UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Confidential BFT State Machine Replication

Robin Vassantlal

Mestrado em Engenharia Informática
Especialização em Arquitetura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Alysson Bessani

2019

Acknowledgments

First of all, I want to thank my advisor, Professor Alysson Bessani. It was during the classes of the professor in the first year of this master course that I was convinced to finish this degree by working on a master thesis with the professor. I was fortunate that the professor accepted me as his pupil, knowing I am not a good writer nor a good speaker, and continuously motivating me to go beyond.

I also want to thank my colleagues and friends for helping me and showing what can I improve so that I can excel. Additionally, I want to thank João Sousa for helping with BFT-SMaRt.

Very importantly, I want to thank my family, especially my parents, for doing all they could to help me easly adapt to the new education system. Without it, I was not even able to continue my studies.

Finally, I gratefully acknowledge the financial support from Fundação para a Ciências e Tecnologia (FCT) through the project Abyss (PTDC/EEI-SCR/1741/2014) and LASIGE (UID/CEC/00408/2019).

Dedico esta tese a todos os professores que me influenciaram na minha trajetória.

Resumo

Vivemos num mundo digital. Cada vez mais dependemos de serviços que empregam replicação para oferecer disponibilidade 24 horas por dia, 7 dias por semana. A Replicação por Máquinas de Estado (RME) é um modelo de replicação em que múltiplos processos/réplicas deterministicamente e em paralelo executam a mesma sequência de comandos para produzir um mesmo resultado [36]. Esta abordagem permite implementar serviços tolerantes a faltas que se mantém corretos e disponíveis mesmo que uma fração de replicas falhem.

Na RME as réplicas executam um protocolo de consenso (e.g., Paxos [24], RAFT [31]) para concordarem sobre a sequência dos comandos que executam. Na literatura podem ser encontrados vários algoritmos que permitem implementar RME Tolerante a Faltas Bizantinas (TFB) (e.g., PBFT [9]). Esses algoritmos asseguram a execução correta do consenso mesmo quando processos faltosos desviam arbitrariamente da sua especificação [25].

A RME também pode ser usado para implementar serviços tolerantes a intrusão [16]. No entanto, proteger a informação guardada no sistema replicado é um problema difícil de resolver, porque existem múltiplas réplicas com os mesmos dados, o que aumenta a superfície de ataque do adversário. Por exemplo, um adversário que consiga comprometer uma das máquinas que implementa um serviço de armazenamento de chave-valor, vai conseguir aceder todas as chaves e os valores associados a elas.

Contudo, existem várias técnicas que resolvem o problema de manter a confidencialidade na RME. *Secret sharing* [7, 39] é uma delas. Esta técnica protege a confidencialidade de dados dividindo os em pedaços/*shares*, de tal forma que os dados só podem ser recuperados através da combinação de um número mínimo necessário de *shares*. Logo, cada servidor, em vez de guardar os dados, guarda um *share* deles (e.g., no DepSpace, apesar de globalmente o serviço armazenar tuplos, localmente cada réplica guarda os *shares* desses tuplos). Assim, se um adversário comprometer uma réplica, só vai conseguir obter o *share* do dado dessa réplica, o que não é suficiente para sua reconstrução.

Todavia, usar apenas *secret sharing* não é suficiente para proteger o serviço contra um adversário móvel [32]. Este adversário salta de uma réplica para outra como um vírus e pode, eventualmente, juntar o número mínimo de *shares* necessários para reconstruir os dados. Um esquema de *proactive secret sharing* [20, 32, 38] pode ser empregue para proteger o serviço contra este tipo de adversário. Este esquema periodicamente renova os *shares* guardados pelos servidores e de seguida apaga os *shares* anteriores. Portanto, é possível proteger o estado do serviço se os servidores forem reiniciados e os *shares* forem renovados antes que o adversário recolha *shares* suficientes.

Existem vários sistemas que oferecem confidencialidade dos dados guardados no serviço, usado para tal o esquema de *secret sharing* (e.g., COCA [43], CODEX [29], DepSpace [6], Belisarius [33] e

sAVSS [3]). Porém, apenas o COCA e o CODEX são resilientes contra o adversário móvel, através da renovação periódica dos *shares* de suas chaves de serviço.

Os protocolos de renovação e recuperação tipicamente usam polinómios aleatórios [20] que permitem renovar e recuperar o *share* sem revelar esse *share* ou o segredo correspondente. Porém, a criação do polinómio e a verificação dos *shares* durante a recuperação é um fator limitante. Por exemplo, não é possível recuperar *shares* de milhares de segredos (e.g., cada *share* correspondente a um tuplo no DepSpace [6]) em tempo razoável sem pré-computar a informação de recuperação previamente [3] (esta informação, além de ter custo de computação, também ocupa espaço de armazenamento nos servidores).

Adicionalmente, de acordo com nosso conhecimento, todos os sistemas replicados práticos que empregam *secret sharing* definem um limite fixo de faltas e membros do sistema durante sua inicialização. Portanto, não existem sistemas eficientes que integram um mecanismo para reconfigurar o sistema durante o seu funcionamento.

Nesta dissertação de mestrado propomos a framework COBRA (COnfidential Byzantine ReplicAtion) para concretização de serviços confidencias baseados em RME. Esta framework suporta a renovação e recuperação de milhares de *shares* eficientemente, bem como a reconfiguração do grupo de servidores e do número de faltas toleradas em tempo de execução.

Ao contrário de sistemas práticos anteriores, onde a solução tinha sido desenhada para resolver um problema específico (e.g., a implementação do COCA integra *secret sharing* para prestar um serviço de autoridade de certificação), a COBRA permite a construção de qualquer tipo de sistema de armazenamento determinista que não requer modificar os dados confidenciais armazenados.

Até onde sabemos, a COBRA é o primeiro middleware para sistemas replicados com confidencialidade que suporta durabilidade, renovação de *shares* (possibilitando rejuvenescimento proativo [10, 41]), recuperação do estado (posterior ao crash) e reconfiguração. Comparado com os sistemas anteriores que integravam apenas algumas dessas capacidades (e.g., COCA [43] e CODEX [29] não tem recuperação dos *shares*, enquanto o DepSpace [6] não tem protocolos para renovação e recuperação de *shares*), o nosso tem todas, inclusive, pela primeira vez, a reconfiguração.

Uma última inovação importante da COBRA é a capacidade de renovar ou recuperar um grande número de *shares* eficientemente. Esquemas anteriores eram ineficientes devido a número elevado de comunicações e custo elevado de verificação dos *shares*. Nosso sistema usa o mesmo polinómio de renovação e recuperação para renovar os *shares* e recuperar o estado, respetivamente, de forma eficiente, e emprega um esquema híbrido de verificação dos *shares*. Este esquema combina um esquema de deteção de *shares* inválidos [18] durante a reconstrução dos polinómios (que pode ser feito de forma rápida) e um esquema de identificação de *shares* inválidos [14] (que é computacionalmente custoso) quando deteta problemas.

O desenho da COBRA foi integrado na biblioteca de replicação BFT-SMaRt [1] e, para validar a solução, foram implementados um serviço de armazenamento de chave-valor tolerante a intrusão e um espaço de tuplos seguro.

Nós avaliamos experimentalmente a solução para medir o impacto de integração de uma camada de confidencialidade num sistema RME. Os resultados mostram, como o esperado, que a integração de *secret sharing* tem um impacto negativo no desempenho do sistema, mas que este impacto se dilui quando os dados manipulados são grandes. No entanto, com a integração do esquema híbrido de interpolação de

polinómios corretos (usando pontos válidos), diminuímos em 100 vezes o tempo de recuperação de um estado composto por 1024 *shares* em relação ao protocolo mais completo existente na literatura.

O trabalho desenvolvido neste projeto resultou numa publicação a ser apresentada no INForum 2019, na track "Segurança de Sistemas de Computadores e Comunicações" [35].

Palavras-chave: Replicação, Tolerância a Faltas Bizantinas, Tolerância a Intrusões, *Secret Sharing*, Confidencialidade

Abstract

State Machine Replication (SMR) is a classical approach to implement consistent fault-tolerant services. This approach can also be used to implement intrusion tolerant services - which maintain integrity and availability in the presence of Byzantine faults. However, in order to guarantee confidentiality, a proactive secret sharing scheme must be used to split data into shares to be distributed between servers. These shares should be periodically renewed to protect the secrecy of the data against a mobile adversary. Current share renewing protocols are designed to renew shares of a single secret, and thus are very inefficient when applied to systems that stored a large number of shares.

We propose a generic framework called COBRA that successfully tackles such scalability problem. COBRA integrates Feldman's secret sharing scheme to split data into shares with Herzberg et al. proactive scheme to periodically renew shares and allow the recovery of a server's state, thus providing protection against a mobile adversary. Furthermore, the framework allows the reconfiguration of the system at runtime.

We mitigated the impact of recovering a large state composed of multiple shares by proposing an Optimistic Polynomial Interpolation (OPI) scheme that integrates Harn and Lin detection scheme to mitigate the cost of verifying Feldman's commitments in fault-free scenarios.

We implemented COBRA on top of BFT-SMaRt and evaluated it to understand the impact of integrating secret sharing into BFT SMR. As expected, the results show that integration of secret sharing layer has a negative impact on the system. However, by employing the OPI scheme, we significantly improved the latency of a share recovery and renew protocols.

Keywords: Replication, Byzantine Fault Tolerance, Intrusion Tolerance, Secret Sharing, Confidentiality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's world has gone digital. We depend on many online services that employ replication to be available 24 hours, 7 days a week. State Machine Replication (SMR) is a replication model, in which multiple processes/replicas deterministically execute the same sequence of commands to produce the same output [36]. This approach allows implementing a fault-tolerant service that maintains integrity and availability even if a fraction of its replicas fail.

The required agreement on the sequence of commands to be executed is achieved by running a consensus protocol (e.g., Paxos [24], RAFT [31]). Several algorithms can be found in the literature that allow implementing Byzantine Fault-Tolerant SMR (e.g., PBFT [9]). These algorithms ensure correct execution of consensus under the Byzantine fault model, in which a faulty process can arbitrarily deviate from its specification [25].

However, securing the confidentiality of the information stored in this replicated environment is a hard problem because there are multiple machines with the same data, in clear text, that an adversary can attack. For example, an adversary that compromises a single server that implements a key-value store service, can access all the keys and associated values kept in the system.

Nonetheless, there exists several techniques that can solve the problem of maintaining confidentiality in SMR. Secret sharing [7, 39] is one of them. This technique protects the confidentiality of the data by dividing it into pieces/shares in such a way that the data can only be reconstructed by combining a minimum number of shares. Hence, each server, instead of storing data in clear text, stores a share of the data. Therefore, if an adversary compromises a server, it will only have access to a share of the data, which is not enough to reconstruct it.

However, employing only secret sharing is not enough to protect the service against a mobile adversary [32]. Such adversary can jump from one server to another like a virus and eventually collects the minimum number of shares required to reconstruct the data. A proactive secret sharing scheme [20, 32, 38] can be employed to protect against this type of adversary. This scheme periodically renews shares stored by the servers and deletes previous ones. Thus, if servers are rebooted and shares are renewed before the adversary collects enough shares (in a period we call an epoch), the stored data will be secure. This means that even if the adversary collected some shares before a renewal, these shares are not useful as they can not be combined with the renewed ones.

1.1 Motivation

There are several replicated systems that offer confidentiality of stored data through the use of a secret sharing scheme [39] (e.g., COCA [43], CODEX [29], DepSpace [6], Belisarius [33], and sAVSS [3]). However, among these, only COCA and CODEX are resilient against a mobile adversary, by enabling the periodic renewal of shares.

Protocols to renew and recover shares use a helper polynomial [20] that allows securely renewing and restoring a share without revealing the secret. However, the creation of such polynomial and the verification of shares during recovery is a limiting factor in terms of performance. For example, recovering shares of millions of secrets (e.g., each one corresponding to a tuple in DepSpace [6]) is not feasible in a reasonable amount of time without pre-computing recovery information [3].

Additionally, to the best of our knowledge, all practical systems that employ secret sharing define a fixed number of tolerated faults and membership during its setup. Thus, there are no practical systems that offer a mechanism to reconfigure their replica set at runtime.

1.2 Goals

In this thesis, we aim to design, implement and evaluate a confidentiality layer for BFT SMR that supports replica recovery and group reconfiguration. Moreover, we want to make it scalable to a large confidential state. These goals can be broken in four mains objectives.

First, we want to study proactive secret sharing schemes and state-of-the-art protocols and systems that provide confidentiality in replicated systems. Through this study we want to identify scalability problems in share renewal and recovery protocols.

Second, we want to design a scalable proactive secret sharing framework for BFT SMR. To achieve this goal, we want to improve previous schemes by solving the identified problems.

Third, we want to integrate our solution to BFT-SMaRt [1,5], a well-known BFT SMR library maintained by LaSIGE/FCUL. This integration will allow the implementation of fault- and intrusion-tolerant services in practice.

Finally, we want to validate the devised prototype by implementing some example services on top of it and evaluate experimentally the overhead of providing confidentiality to BFT SMR systems.

1.3 Contributions

In this master thesis, we propose a framework called COBRA (COnfidential Byzantine ReplicAtion) that successfully tackles the problem of scaling a proactive secret sharing scheme to renew and recover millions of shares efficiently, supporting also a reconfiguration protocol that allows changing the membership and number of maximum tolerated faults at runtime. Compared with existing systems, COBRA brings the following innovations:

- COBRA is a *generic framework* that allows the development of different storage services on top of it, except the services that requires to modify stored data;

- COBRA is the first system to support durability, share renewal (enabling Proactive and Reactive Rejuvenations [10, 41]), state recovery (after a crash), and reconfiguration for confidential state machine replication;
- Lastly, COBRA is the first system to allow the renewal and recovery of a large number of secret shares efficiently.

Moreover, we integrate COBRA into BFT-SMaRt [1] and implement an intrusion-tolerant key-value store and a distributed tuple space [6] on top of it.

Finally, we evaluate COBRA from a theoretical and experimental point of view to show the cost of integrating confidentiality into BFT SMR.

1.4 Publication

The work described in this dissertation resulted in the publication of a paper at INForum 2019, in the "Security of computers and communications systems" track [35].

1.5 Structure of the document

This document is organised as follows:

- Chapter 2 presents background and related work. The chapter starts by describing the SMR model and some protocols that implement that model. Next, it describes the evolution of secret sharing from simple share distribution protocol to a protocol that moves shares from one group of shareholders to another. In the end, we discuss BFT systems that provide confidentiality and their limitations.
- Chapter 3 describes the design of COBRA. First, we define the system model and assumptions that we make. Next, we discuss the properties that COBRA ensures. The remaining of the chapter describes the schemes and protocols that allows storing confidential data in a BFT SMR system.
- Chapter 4 gives details about the implementation of COBRA. It describes the implementation of a standalone library that isolates computation related to secret sharing. Next, it describes the integration of COBRA into BFT-SMaRt. We conclude the chapter by discussing two applications implemented on top of COBRA.
- Chapter 5 evaluates COBRA. The chapter starts by presenting a theoretical and practical analysis of the data protection scheme and presenting the cost of all sub-protocols implemented in COBRA. In the end, it shows the impact of integrating confidentiality in a BFT SMR system.
- Chapter 6 concludes the thesis and discusses the future works for improving COBRA.

Chapter 2

Background and Related Work

In this chapter, we begin by recalling the concept of State Machine Replication (SMR). After that, we describe PBFT [9], the first efficient Byzantine Fault Tolerant (BFT) agreement protocol and give a description of the BFT-SMaRt library [1]. Next, we describe the evolution of secret sharing from basic Shamir's scheme [39] to a protocol that securely allows to move shares from one group of shareholders to another. We conclude the chapter by giving examples of replicated systems that provide confidentiality.

2.1 State Machine Replication

The State Machine Replication (SMR) approach [23, 36] allows the implementation of consistent and fault-tolerant services by replicating the same computation into multiple processes (replicas). In this model, clients issue commands to all the replicas, which execute them in a deterministic and coordinated way to produce a replay.

To maintain a consistent state, any implementation of the SMR model requires [36]:

- Same initial state: all correct replicas start on the same state;
- Determinism: all correct replicas receiving the same input on the same state produce the same output and resulting state;
- Coordination: all correct replicas process the same sequence of commands.

First two requirements are easy to ensure, but coordination requires the implementations of a total order multicast primitive [17].

Systems that implement the SMR model must also satisfy the following properties of safety and liveness:

- Safety: all correct replicas execute the same sequence of operations;
- Liveness: all correct clients operations are executed.

The safety property allows the implementation of strongly consistent services, satisfying the consistency property known as Linearizability [19].

The systems that implement SMR model typically consider that processes/replicas fail either by crashing or by presenting an arbitrary behaviour:

- Fail-stop fault model: a faulty (crashed) process halts and does not execute any further operations during the system execution;
- Byzantine fault model: a faulty process may behave arbitrary, i.e., deviating from its algorithm and taking any action.

In terms of the synchrony of the system, there are three relevant system models to consider for practical SMR protocols: asynchronous [15, 27], synchronous [27], and partially synchronous [13].

As previously mentioned, the SMR approach requires the implementation of a total order primitive, which at its core solves the classical problem of consensus [17]. Fischer et al. [15] proved that it is not possible to solve consensus in an asynchronous system model with process crashes, but it is possible in a partially synchronous system model [13], by ensuring the safety property in the absence of synchrony.

There have been many works that have produced several protocols to order requests in the partially synchronous system model. Viewstamped Replication [30], Paxos [24], and RAFT [31] are examples of the protocols that solve consensus for Crash Fault-Tolerant (CFT) SMR systems. PBFT [9] is the first Byzantine Fault-Tolerant (BFT) protocol that achieved a performance similar to its CFT counterparts.

2.1.1 PBFT

PBFT [9] is a SMR protocol that solves consensus in the partially synchronous system model. To tolerate t Byzantine faults, the algorithm requires $3t + 1$ replicas in the system.

This protocol is leader based. The leader assigns a sequence number to requests, while other replicas check these numbers for consistency and monitor the leader to detect when it stops or misbehaves so that one of the other replicas can overtake the leadership.

Figure 2.1 shows the messages exchanged between replicas in PBFT [9] to order a request when the leader is correct and the system is under a synchrony period.

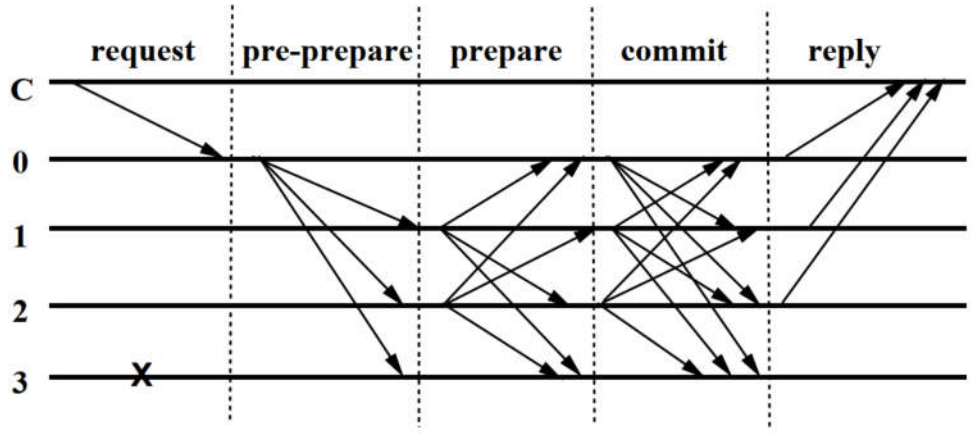

Figure 2.1: PBFT normal case protocol [9]. Process 3 is faulty and does not send messages.

The protocol, in a typical case, works as follows:

1. A client sends a request m to the leader;
2. The leader replica then sends a PRE-PREPARE message to all replicas assigning a sequence number i to m;

3. A replica accepts a PRE-PREPARE message if the leader proposal is good, i.e., the authenticity of m is verified and no other PRE-PREPARE message was accepted for the sequence number i;

4. When a replica accepts a PRE-PREPARE message, it sends a PREPARE message with m and i to all servers;

5. When a server receives $\lceil \frac{n+t}{2} \rceil$ PREPARE messages with the same m and i, it marks m as prepared and sends a COMMIT message with m and i to all servers;

6. When a server receives $\lceil \frac{n+t}{2} \rceil$ COMMIT messages with the same m and i, it commits m, i.e., it accepts that request m is the i-th request to be executed;

7. Once a server executed all requests with a sequence number lower than i, it executes m and sends a replay to the client;

8. The client waits for $t + 1$ matching replies for the request and completes the operation.

The algorithm also has a view change protocol. This protocol allows ensuring liveness when the leader fails. Replicas move from current view to next after changing the leader. View change is triggered by timeouts at each replica. Thus, each replica maintains a timer, which is started when it receives a request and stopped when the request is committed.

2.1.2 BFT-SMaRt

BFT-SMaRt [1] is an open-source Java-based library that implements a robust Byzantine fault-tolerant State Machine Replication algorithm that is similar to PBFT [5]. This library allows the implementation of dependable services and also can be extended to implement new protocols that require the BFT SMR model.

The development of this library follows five principles [5]: tunable fault model, simplicity, modularity, simple and extensible API, and multi-core awareness.

The BFT-SMaRt requires a partially synchronous system model to ensure liveness and reliable authenticated point-to-point links between processes for communication, which are provided by SSL/TLS channels.

To send ordered request, clients call the method $invokeOrdered$ on the Service Proxy object, which in turn sends a request to all the servers in the current view. This request will trigger the execution of a sequence of consensus instances in the servers to decide the order of execution [40].

Additionally, BFT-SMaRt provides a state transfer protocol and a reconfiguration protocol. The former allows a slow or recovering replica to obtain the current state using other replicas states, transferred collaboratively [4]. In case of more than t replicas are down, the use of durability techniques, like logging and checkpointing, allows the state recovery of the whole system. Naively implementing logging and checkpointing decreases the system performance, so BFT-SMaRt uses parallel logging and sequential checkpointing [4] to achieve high throughput while ensuring the durability.

The reconfiguration protocol allows administrators to increase or decrease the size of the system in runtime by adding or removing replicas. To accomplish this, BFT-SMaRt uses a View Manager, a unique, trusted, type of client that is managed by system administrators.

2.2 Secret Sharing

Secret Sharing (SS) is a technique used to split a secret D into n pieces, also called shares. This technique ensures that at least $t+1$ shares out of n are needed to reconstruct D. Additionally, it ensures that having t or fewer shares reveals nothing about the secret. This type of scheme is called a (t, n)-threshold scheme.

When a replicated system employs this kind of scheme, values n and t can be viewed as the total number of servers and the number of faults that the system tolerates, respectively.

In this type of scheme, there are three roles (Figure 2.2): the dealer, the shareholders and the combiner. Typically, the dealer is also the combiner.

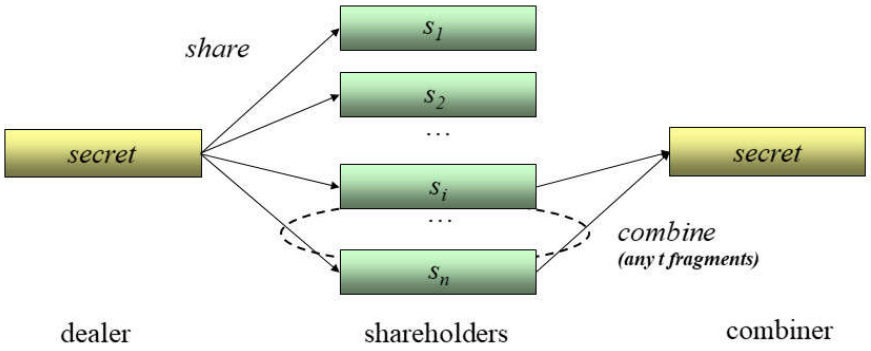

Figure 2.2: Secret sharing scheme.

The dealer has the task of generating n shares of the secret D and distribute those shares among n shareholders. The shareholders receive and store their respective share. The combiner collects $t + 1$ shares from distinct shareholders and reconstructs the secret D.

2.2.1 Basic Scheme

Secret Sharing was invented independently by Adi Shamir [39] and George Blakley [7] in 1979. The Blakley scheme encodes the secret in a coordinate of the point intersection of nonparallel k-dimensional hyperplanes. Thus each share defines a plan. The Shamir scheme, used in this work, encodes the secret D as the constant term of the polynomial P of degree t, $P(x) = D + a_1x^1 + ... + a_tx^t$, with other t coefficients, $a_1, ..., a_t$, chosen randomly. The share of the shareholder i is a point on the polynomial P, $P(i)$, where i is the unique identifier of the shareholder, which must be different from 0. The secret is recovered by interpolating $t + 1$ distinct shares $(x_i, y_i) = (i, P(i))$ to obtain the polynomial P and computing $P(0) = D$. Such interpolation can be done by using the Lagrange method, as shown in Equations 2.1 and 2.2.

$$P(x) = \sum_{j=1}^{t+1} y_j l_j(x) \tag{2.1}$$

$$l_j(x) = \prod_{\substack{1 \le m \le t+1 \\ m \neq j}} \frac{x - x_m}{x_j - x_m} \tag{2.2}$$

2.2.2 Verifiable Secret Sharing

A basic SS scheme does not have a mechanism to detect whether the shares were corrupted or are related with the same secret. Thus, a malicious dealer can distribute shares from different polynomials to the different shareholders, or some of the shareholders can be faulty and send corrupted shares to the combiner. In both cases, the combiner may collect some of the corrupted shares in the set of $t+1$ shares and reconstruct an incorrect secret.

A Verifiable Secret Sharing (VSS) scheme prevents the shareholders from storing corrupted shares and allows the combiner to collect valid shares of the same secret.

VSS schemes can be grouped into two categories: interactive and non-interactive. Interactive schemes require communication to verify the shares while non-interactive schemes allow each party to do local verification of the shares.

Non-interactive VSS schemes can be easily implemented since they do not require communication between the prover and the verifier. Hence, next, we give examples of some of the non-interactive VSS schemes.

VSS schemes based on quorums. The quorum based approach uses the shares itself to detect and identify corrupted shares, without the need for any additional information.

Harn and Lin [18] present a scheme that uses a set C of more than $t+1$ shares to detect and identify corrupted shares. Since it requires more than one share, only the combiner (not the shareholders) can do such verification.

The combiner uses set C to reconstruct the polynomial P of the secret. If P has degree t (the threshold used to share the secret), it means there are no corrupted shares in C, and the secret is $P(0)$. Else, C contains corrupted shares. The scheme requires the use of distinct $2t+2$ shares in C to identify corrupted shares and the real secret. This procedure requires the reconstruction of $\binom{2t+2}{t+1}$ possible secrets. The reconstructions are grouped into sets, and each set contains the set of shares that reconstructs the same possible secret. The real secret is the value reconstructed by set S of reconstructions that has a maximum size. All the shares that were not in the set S are corrupted.

VSS schemes based on extra information. The VSS schemes based on extra information attach additional information to the shares. This information is used by the shareholders and the combiner to detect and identify corrupted shares.

The works [14, 22, 34] are examples of VSS schemes that append additional information in the form of commitments.

Feldman's [14] and Pedersen's [34] schemes create a vector of commitments for each polynomial used in Shamir SS, one commitment per coefficient. All the shares of the secret are verified using the same vector of commitments. Feldman's [14] scheme is perfectly binding and computationally hiding, while Pedersen's [34] scheme is computationally binding and perfectly hiding. The perfectly binding property means there is only one possible value that can generate the corresponding commitment. The perfectly hiding property means it is not possible to find the value that originated the commitment. The computational version of binding and hiding means that, if the discrete logarithm problem is hard, then it is very hard to find the value that originated the commitments.

The goal of Kate et al. [22] scheme is to reduce communication and storage cost, which is achieved by computing constant size verification information (i.e., independent of the number of coefficients/faults). However, the computation cost is superior compared to Feldman's scheme.

Since we use Feldman's [14] VSS in COBRA, we describe next how this scheme works.

Feldman's scheme extends the Shamir's scheme [39] by adding two algorithms: $commit$ and $verify$. The $commit$ is used by the dealer to generate a vector of commitments for the Shamir's polynomial P, while $verify$ is used by the shareholders and the combiner to verify if a share was created using the same polynomial P from which the commitments were generated.

This scheme uses numbers g and p as public system parameters, where:

- g is a generator of cyclic group G;
- G is a subgroup of $\mathbb{Z}_p^*$ of prime order q.
- $p = lq + 1$, p is also prime, and l is a positive number;

The dealer generates the vector of commitments $[c_0, c_1, ..., c_t]$ for the polynomial P as follow:

$$c_i = \begin{cases} g^D, \text{if } i = 0 \\ g^{a_i}, \text{else if } i = 1..t \end{cases} \tag{2.3}$$

The shareholders and the combiner verify if share s_i is a point on the polynomial P by checking if the following equality is true:

$$\begin{aligned} g^{s_i} &= g^{P(i)} \\ &= g^{D+a_1 i^1 + ... + a_t i^t} \\ &= g^D * (g^{a_1})^{i^1} * ... * (g^{a_t})^{i^t} \\ &= \prod_{j=0}^{t} (c_j)^{i^j} \end{aligned} \tag{2.4}$$

2.2.3 Proactive Secret Sharing

SS schemes maintain the secrecy of the data if throughout the entire lifetime of the secret the adversary is restricted to compromise at most t out of n shareholders. However, this is not generally true in the real world, where a malicious attacker can compromise more than t servers in a long-lived system.

Proactive Secret Sharing (PSS) schemes allow the renewal of the shares without revealing the secret, thus periodically shareholders renew their shares, making their old shares (possibly in the hands of the attacker) useless.

In this approach, the system moves from epoch e to epoch $e+1$ after executing the renewal protocol. The secrecy of the data is maintained if the duration of the e is less than the time required for an attacker to get $t + 1$ shares from the epoch e.

Herzberg et al. [20] introduce a practical share renewal protocol, for the synchronous model, to allow each shareholder to compute their new share as a function of its corresponding old share. This work also introduces a recovery protocol to allow a shareholder to recover its share using other shareholders shares, without compromising the secrecy of the data or the other shareholders' shares.

The share renewal is done using a random polynomial Q of degree t, such that $Q(0) = 0$. The i's renewed share $P'(i)$ is then $P'(i) = P(i) + Q(i)$. This scheme does not modify the secret because $Q(0) = 0$, thus $P'(0) = (P + Q)(0) = P(0) = D$, but changes i's share since $P'(i) = (P + Q)(i) \neq P(i)$.

Shareholder j's share recovery is done using a random polynomial R_j of degree t, such that $R_j(j) = 0$. Each shareholder i, that has its share $P(i)$, sends $P(i) + R_j(i)$ to j, which reconstructs the polynomial $P + R_j$ and evaluates it at j to obtain its share. The polynomial $P + R_j$ does not expose any information about the secret, because $R(0) \neq 0$, thus $(P + R_j)(0) \neq D$, and the recovered share is of the polynomial P, because $R_j(j) = 0$, therefore $(P + R_j)(j) = P(j)$.

This scheme requires polynomials Q and R_j to be created in a distributed way. This way, all the shareholders use the points of the same polynomials. The idea is that each shareholder creates its local polynomial, and the final polynomial is the sum of the local polynomials of the correct shareholders [20].

2.2.4 MPSS - Mobile Proactive Secret Sharing

Herzberg et al. protocol, just described, allows the renewal of the shares within the same group of shareholders in a synchronous system model. Schultz et al. [38] introduce a generic protocol for share renewal for the asynchronous Byzantine system model. Beside share renewal, this protocol also enables the transfer of shares from one group of shareholders (old group) to another (new group) during the renewal, enabling the reconfiguration of group of shareholders. Additionally, it allows modification of the number of faults tolerated by the system without restarting it.

In terms of cryptographic assumptions, each server has two pairs of public-private keys. One pair is used for signing and authenticating the messages, and the other pair is used to encrypt and decrypt the content of the messages. The authors use forward-secure encryption [8], which allows each server to change their private keys. This way if the adversary records all the messages and in the future compromises private keys of some server, it will be unable to decrypt old recorded messages.

During the share renewal, the system can tolerate at most $2t$ Byzantine faults, t faults in each group of shareholders. To renew and move the shares without compromising the secret or the shares, the authors combine Herzberg et al.'s [20] renewal and recovery scheme into a single protocol. Thus during each renewal, old shareholders generate $n+1$ polynomials (1 renewal polynomial and n recovery polynomials - one for each shareholder of the new group). Additionally, this protocol uses VSS (e.g., Feldman's [14] or Pedersen [34]) to allow each shareholder to verify shares sent by other shareholders.

MPSS [38] renewal protocol is divided into two phases. In the first one, executed between shareholders of the old group, $n + 1$ random polynomials are created. In the second phase, the shareholders of the old group transfer shares to the shareholders of the new group.

Random polynomials are generated in a distributed way. Each shareholder i creates a local polynomial Q_i and $R_{i,k}$ and exchange between them proposals containing points on those local polynomials. They agree (using PBFT [9]) on set S of proposals that are valid for all the correct shareholders. The polynomials Q and R_k is a sum of the polynomials contained in S.

To transfer the shares, each old shareholder i, generate n transfer shares $T_k(i) = P(i)+Q(i)+R_k(i)$ (one for each new shareholder k) and sends a message to all the new shareholder containing all the $T_k(i)$ (each $T_k(i)$ encrypted for new shareholder k) and the commitments of Q, R_k, and P. Each new

shareholder k use $t + 1$ valid points of T_k (verified using commitments) to interpolate the polynomial $T_k = P + Q + R_k$ and computes $T_k(k)$ to obtain its renewed share.

2.3 Related Work: Confidentiality in BFT SMR

Several systems add confidentiality to the Byzantine Fault-Tolerant State Machine Replication (BFT SMR) model through careful design of the system or using some mechanism to obfuscate the data.

The works of Yin et al. [42] and Duan and Zhang [12] extends the SMR model to support a new architecture to provide confidentiality of the data and still allow the execution of arbitrary operations on the data. The architecture separates the agreement phase from the execution phase into a different cluster of machines. The agreement cluster runs a BFT consensus protocol to order requests and assign a unique sequence number, while the execution cluster, isolated in a separated network, executes the requests in the order defined by the sequence number. This separation requires only $2g+1$ machines in the execution layer to tolerate g faults, but still, requires $3f + 1$ machines in the agreement layer to tolerate up to f Byzantine faults. A layer of privacy firewall is inserted between agreement and execution layers to filter incorrect responses and to ensure the confidentiality.

The separation of the agreement from executions requires high infrastructure cost and the network must be carefully designed to isolate the execution cluster (that must be accessible only through the privacy firewall) since these machines hold the data in clear.

COCA [43], CODEX [29], DepSpace [6], Belisarius [33], and sAVSS [3] are examples of the systems that use the SS to ensure confidentiality by storing the shares of the data into each server.

COCA [43] is an online certification authority, and CODEX [29] uses COCA's [43] protocols to implement a distributed service for storage and dissemination of secrets. Both services use Byzantine quorum replication [28] instead of SMR, but the technique used to protect the data is applicable to this model. These systems have a global public/private key pair as a part of the service that they implement. The servers use the SS mechanism and proactive share renewal to distribute shares of the service private key among them and to periodically renew those shares, thus providing intrusion tolerance and resistance against a mobile adversary [32]. Each server uses its share of service private key to generate a partial cryptographic result, and those partial results are combined to create a final result as if it was generated using the private key. In COCA, the service private key is used to sign certificates which can be verified using the service public key. In CODEX, the clients use the service public key to encrypt and store their secret keys in the system. There secrets can be recovered using the blinding [11] technique without compromising the secrecy of the client's secret keys while being decrypted using service private key shares. Interestingly, in both services, the share renewal is not a bottleneck, because there is only one secret whose shares are to be renewed.

Unlike COCA [43] and CODEX [29], DepSpace [6], Belisarius [33], and sAVSS [3] employ SS to encode each data entry (e.g., tuple, key-value pair) stored in these systems.

DepSpace [6] is a dependable tuple space based on BFT SMR. The authors use SS to divide a tuple into shares and the concept of fingerprint to allow each server to search its share of the tuple given the tuple's template. The system architecture (Figure 2.3) consists of a series of integrated layers that enforce each one of the dependability attributes. The bottom layer in each side (client and server) is a

replication layer, which guarantees that failures of up to t servers will not affect the reliability, availability and integrity of the system. In the confidentiality layer, a publicly verifiable secret sharing scheme [37], which combines the Shamir's secret sharing with extra information that allows the verification of shares, is used to encode tuples as shares. The access control layer uses credentials to define which clients can access the tuple space and what type of operations they can execute. The policy enforcement layer in the server-side uses the identifier of the client, the operation, its arguments, and the tuples currently in the tuple space to decide if an operation is approved or denied. Each tuple space has a single access policy defined during the system setup by the administrator. One of the optimizations applied to DepSpace is that during tuples insertions, each server stores all the shares (shares of the other servers are encrypted). Servers only decrypt their share and generate the proof when the first read request arrives.

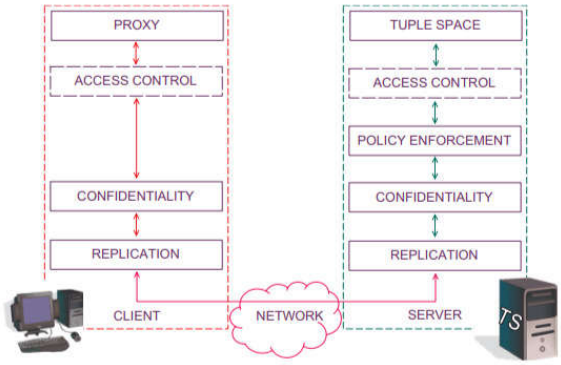

Figure 2.3: DepSpace architecture [6].

Belisarius [33] is a confidential BFT storage system. This system is composed of three main components: the client-side confidentiality handler, the server-side obfuscated data manipulator and the BFT communication protocol. Belisarius uses Shamir's secret sharing to ensure confidentiality of the data. The clients divide the data into shares and uses total order multicast to send shares to the servers. This system uses a quorum based approach (explained in Section 2.2.2) to identify corrupted shares and correct reconstructed data, instead of appending extra information to the shares.

sAVSS [3] is a recent framework that transforms a verifiable secret sharing (VSS) scheme into an asynchronous VSS with constant storage and communication overhead. This framework uses Kate et al. [22] polynomial commitments scheme to allow the verification of the shares. The main advantage of this scheme is that it has a constant cost of transmission and takes constant storage space, independent of the number of faults the system tolerates. Additionally, it modifies Herzberg et al. [20] recovery scheme to allow multiple share recovery using the same polynomial. The idea behind the recovery is that the dealer preemptively creates recovery polynomials to recover $3t + 1$ shares. However, instead of generating one recovery polynomial per share, it encodes t shares into each recovery polynomial.

2.4 Discussion

We started this chapter by recalling the concept of state machine replication and describing the BFT-SMaRt library. Next, we show the evolution of secret sharing mechanism from the simplest scheme to a general protocol that allows moving shares from one group of shareholders to another.

System	Replication Approach	Renewal	Recovery	Reconfiguration
COCA [43]	Quorum	Y	N	N
CODEX [29]	Quorum	Y	N	N
DepSpace [6]	SMR	N	N	N
Belisarius [33]	SMR	N	N	N
sAVSS [3]	SMR	N	Y	N
COBRA	SMR	Y	Y	Y

Table 2.1: Replicated systems with confidentiality.

In the last section, we gave examples of systems that provide confidentiality. Table 2.1 summarises the mechanisms that those systems implement. The systems are categorized in a four groups: replication approach used for fault-tolerance; if has renewal mechanism to protect system against mobile adversary; if has recovery protocol to rebuild a server's shares without reconstructing the secret; and if has reconfiguration mechanism to change membership and number of faults that system tolerants at runtime. As can be seen in the table, none of the systems has all of the mechanism implemented by COBRA (see next chapter). Moreover, the few systems that provide renew or recovery does not scale to multiple secrets, for example, COCA and CODEX's renewal protocol is used to renew shares of just one secret. Thus, in those systems the scalability of renewal protocol is not an issue. sAVSS's recovery protocol requires the clients to generate recovery information, and each server has to store that information. Finally, none of the systems integrates a reconfiguration protocol.

Chapter 3

Confidential BFT State Machine Replication

In this chapter, we present the design of the COBRA framework that enables the implementation of confidential secure long-running services based on BFT SMR.

The framework integrates verifiable and proactive secret sharing to protect the confidential data against a mobile adversary and to recover servers. Additionally, we integrate the reconfiguration protocol to reconfigure the system at runtime.

The following sections describe our system model and how we integrated all the mechanisms into the design of COBRA.

3.1 System Model, Assumptions, and Notation

We consider a system with $n \geq 3t + 1$ replicas that tolerate at most t simultaneously Byzantine faults. The system is partially asynchronous and the communication between replicas is done using secure point-to-point channels (implemented using SSL/TLS).

An unlimited number of clients access the service. Each client l has a encryption key $K_{l,i}$ shared with the replica i.

We assume each replica has a public unique identifier $i = 1..n$. The SS protocol uses this identifier to compute shares.

The system evolves in a sequence of epochs. After each share renewal, the system goes from epoch e to epoch $e + 1$. In each epoch, an adversary can corrupt at most t replicas, but in the entire lifetime of the system, an adversary can corrupt more than t replicas.

The system has prime numbers q, p, and generator g for the cyclic group G as public parameters, such that, q divides $p - 1$ and G is a subgroup of $\mathbb{Z}_p^*$ of order q. Moreover, q must be greater than the secret (encoded as number) and n, and the discrete logarithm problem is hard in the group $\mathbb{Z}_p^*$.

All computations from Shamir's scheme [39] are performed on the finite field of order q. The computations related to the Feldman's scheme [14] are done in the cyclic group $\mathbb{Z}_p^*$ using the generator g.

3.2 Service Properties

The SMR model defines that all correct replicas must have the same state, which is deterministically modified by the ordered sequence of commands. However, in our system, this cannot be ensured since different replicas have different shares of the stored data. Therefore, our SMR system globally stores the confidential data D, but locally each server i maintains share s_i of a random key K used to encrypt D. More formally, given a global state $S = \{D_1, ..., D_m\}$ composed of m data entries, each correct server i maintains a state $S_i = \langle O_i, E_i \rangle$ that can be divided in two parts:

- $O_i = \{\langle \overline{D_j}, D_j^e, V_j \rangle : j = 1..m\}$ is the *open data*, represented by a set of tuples containing the information that is replicated in all servers. Each of these tuples contains the following fields:
  - $\overline{D_j}$ is the non-sensitive data associated with the data entry (e.g., the id of a secret [29], the key associated with a confidential value, or the public fingerprints of tuple [6]);
  - D_j^e corresponds to D_j encrypted using a random symmetric key K_j;
  - V_j is vector of commitments for the secret's polynomial used to generate the shares of K_j;
- $E_i = \{s_{i,1}, ..., s_{i,m}\}$ is the *private data*, which contains i's shares for each key K_j.

In addition to ensuring the standard *safety* and *liveness* properties of SMR (see Section 2.1), COBRA also ensures a new safety property designated as *secrecy*.

- *Definition (Secrecy):* A replicated system composed by n servers satisfies *secrecy* (of its state $S = \{D_1, ..., D_m\}$) if no information about any data entry D_j of its state can be obtained by an unauthorised party even with up to t simultaneous Byzantine servers.

In other words, this property ensures that no information about D_j can be obtained if at most t servers *private data* are compromised in an epoch e. The consequence is that an adversary is required to collect distinct $t + 1$ shares of D to have access to it. By using secret sharing and storing different shares in each server, this property directly follows from the fact that obtaining t or less shares of a secret is not enough to reveal the key used to encrypt it.

3.3 COBRA's Architecture

The COBRA's architecture follows the module design presented in Figure 3.1. The framework sits on top of the replication layer. The replication layer provides secure point-to-point connections between replicas and between clients and replicas. Additionally, the replication layer implements a BFT agreement protocol to order requests of clients.

The framework has two main modules, the confidentiality proxy and the confidentiality handler. The confidentiality proxy module, located at the client-side, implements the data protection scheme described in Section 3.5. The client uses this proxy to write and read confidential data to and from the replicas.

The confidentiality handler is located in the replicas. This module is composed of two sub-modules, the polynomial manager, and state manager. The confidentiality handler extracts its share of the data and delivers it to the service, being also responsible to separate the shares from the public data. The

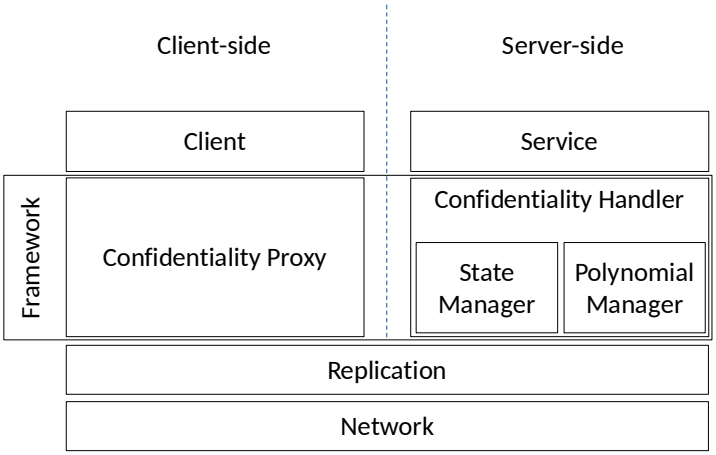

Figure 3.1: Overview of the COBRA's architecture.

polynomial manager implements the distributed polynomial generation protocol described in Section 3.6. Finally, the state manager implements the share renewal and recovery protocols described in Sections 3.7 and 3.8, respectively.

3.4 Optimistic Polynomial Interpolation (OPI)

We implemented a mechanism that uses Harn and Lin detection scheme [18] and Feldman's commitment scheme [14] to verify shares and interpolate a correct polynomial efficiently. Figure 3.2 presents the process followed to interpolate the correct polynomial.

The detection scheme [18] has the priority during the interpolation. This scheme allows detecting invalid shares during the polynomial interpolation using a set C of $t + 2$ points. If the polynomial has degree t, then the polynomial is correct. Else, there were at most t invalid points in C. Next, the mechanism uses the commitment scheme to identify the invalid points and the servers that sent those invalid points. The identification is made using the commitments (as explained in Section 2.2.2). The verification stops when the mechanism identifies set C' of $t + 1$ correct points. The correct polynomial is interpolated using this set of correct points.

The mechanism puts the servers that sent the invalid points in a blacklist so that in the next interpolation the set C will not have points from those servers. If the blacklist already has t servers it means that t corrupted servers were identified and for now on the set C will have only $t + 1$ points.

The execution of this mechanism begins when the combiner or the recovering server collects $2t + 1$ points and commitments and ends by returning the correct interpolated polynomial F of degree t.

Notice that it is not necessary to wait for $2t + 1$ answers to start the fast path of the procedure, only $t + 2$ are sufficient. More points are required only when the detection scheme detects invalid points [18].

In Chapter 5 (of evaluation) we demonstrate the advantages of the use of two schemes together, notably when the combiner or the recovering server interpolates multiple polynomials.

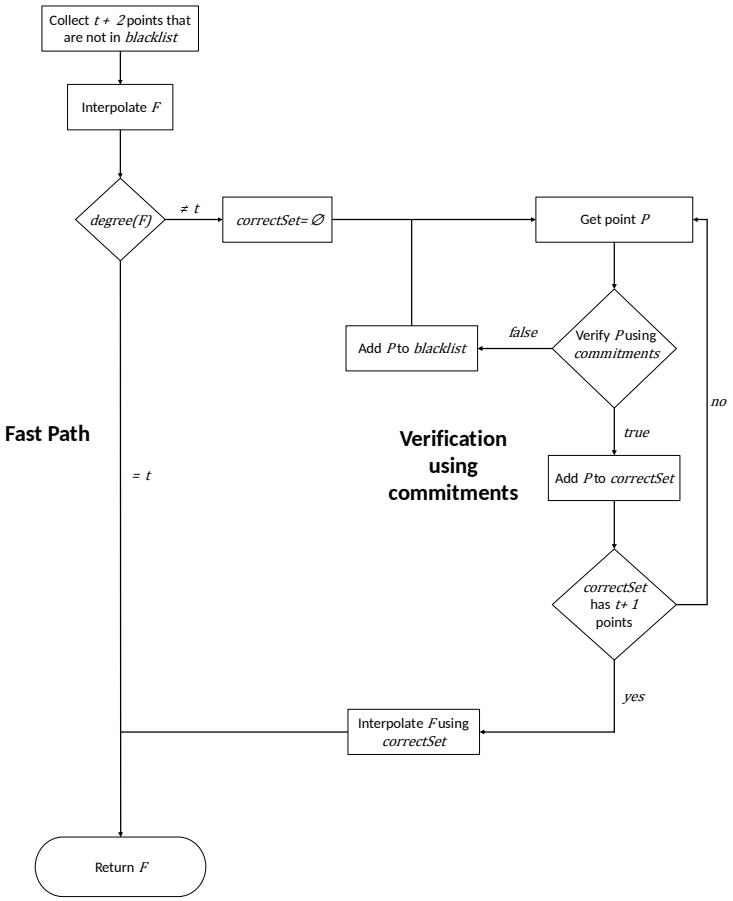

Figure 3.2: Flowchart of polynomial interpolation of degree t.

3.5 Data Protection Scheme

In this section we present the two protocols ($share$ and $combine$) used by the framework to store confidential data into the replicated system securely and to read it back.

To store the data, we use symmetric encryption to encrypt the data and use Shamir's secret sharing scheme [39], augmented with Feldman's commitments [14], to split the key into shares that each server store. Additionally, we use the detection scheme described in previous section to reduce the use of commitments during the secret reconstruction. This scheme will allow detecting corrupted shares during the secret's polynomial interpolation.

3.5.1 The Share Protocol

The $share$ protocol is executed at the client-side of the framework to send confidential data to the service. The sequence diagram presented in Figure 3.3 describes the interactions that happens in the system to write confidential data.

The execution begins when the client l wants to store the confidential data entry D. The confidentiality proxy encrypts D using a random symmetric encryption key K, generating $D^e = Enc(D, K)$. After that, it uses Shamir's scheme [39] to create a polynomial P encoding K as a secret and computes n shares, $s_1, ..., s_n$. Then, the confidentiality proxy generates the vector of commitments c for P using

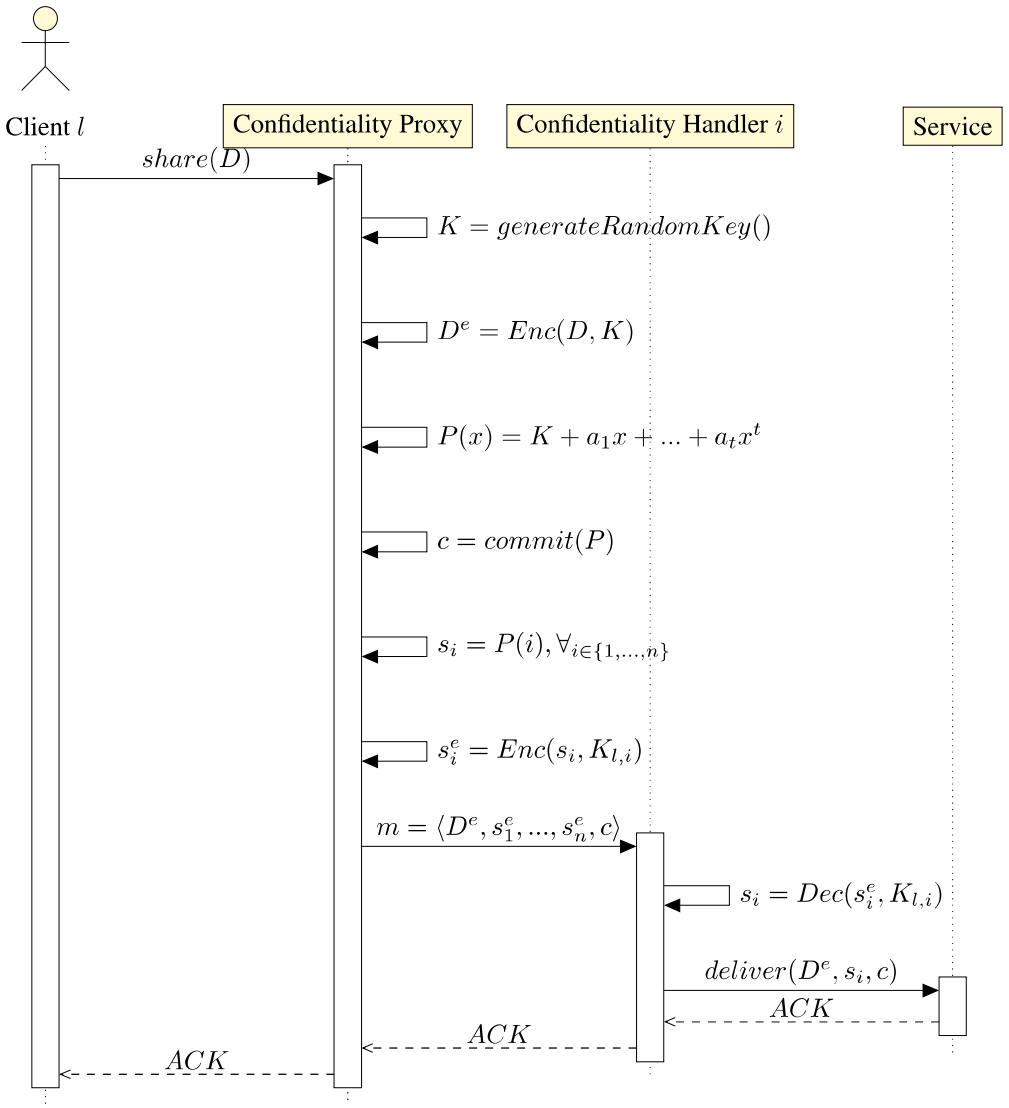

Figure 3.3: Sequence diagram of the *share* protocol.

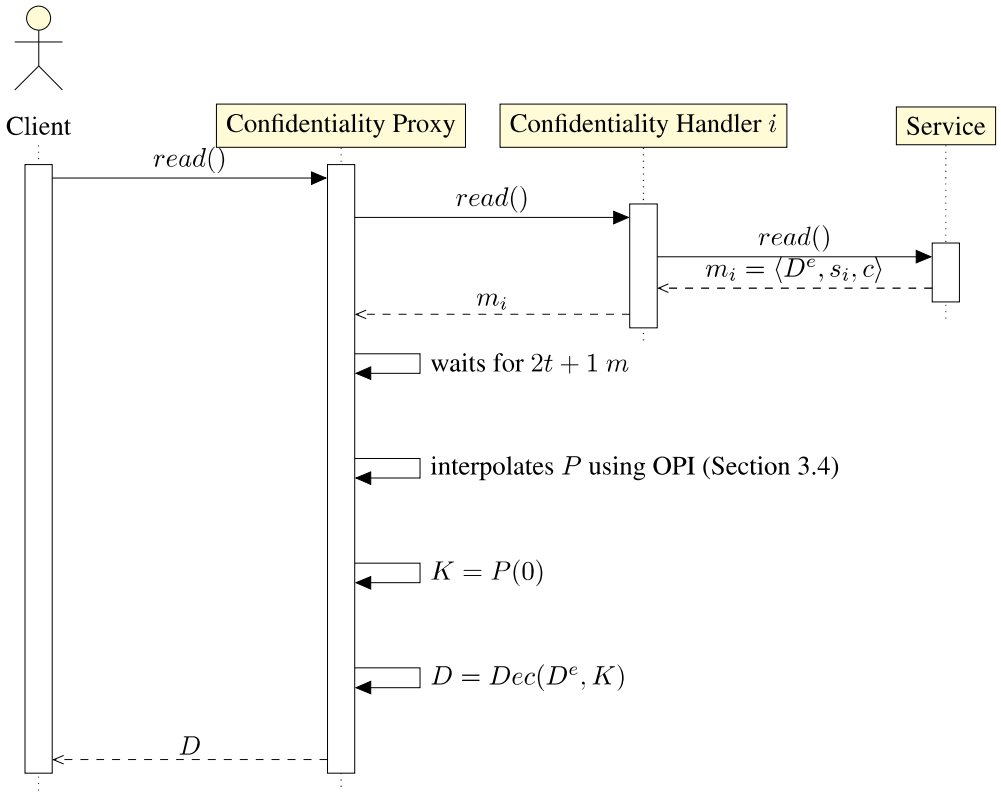

Figure 3.4: Sequence diagram of the *combine* protocol.

Feldman's commitment scheme [14]. In the end, the confidentiality proxy sends a message m to all the shareholders/servers using a total order multicast primitive. The message m contains D^e, all the shares (each share s_i is encrypted using shared encryption key $K_{l,i}$ with the corresponding server i), and the vector of commitments c. At each replica i, the confidentiality handler extracts and decrypts its encrypted share s_i and delivers it with the encrypted data D^e, and the vector of commitments c to the service.

3.5.2 The Combine Protocol

The *combine* protocol is executed at the client-side of the framework to read the confidential data stored in the service. The sequence diagram of Figure 3.4 presents the interactions that happens in the system to read confidential data from the servers.

The confidentiality proxy asks for the encrypted confidential data D^e, its shares, and commitments to the shareholders/servers. Each server sends the requested information using the secure point-to-point channels. The confidentiality proxy collects $2t + 1$ shares from distinct servers and using mechanism described in Section 3.4 interpolates the correct secret's polynomial P. After interpolating P, the confidentiality proxy obtains the key $K = P(0)$ and decrypts D^e to obtain the confidential data D.

3.6 Distributed Polynomial Generation Protocol

The distributed polynomial generation protocol allows servers to create a global random polynomial F of degree t with some fixed point (x, y) such that $F(x) = y$. At the end of the protocol, each server i will receive its point $F(i)$ and a vector of Feldman's commitments for the generated polynomial F.

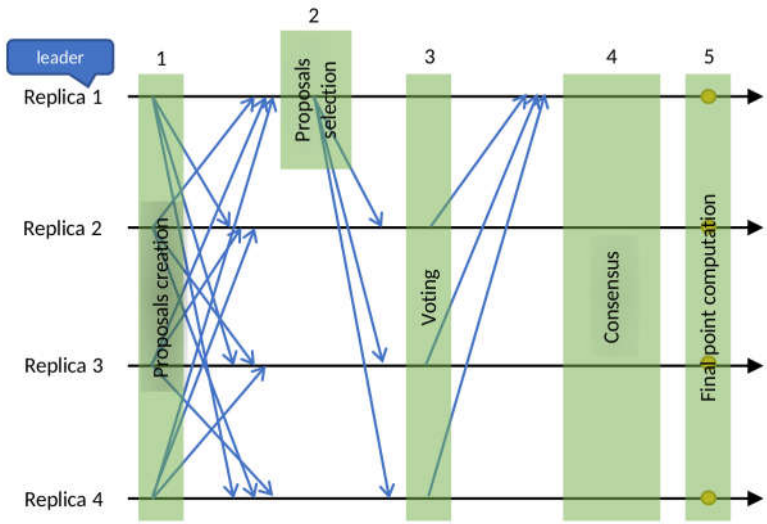

Figure 3.5: Distributed polynomial generation in the system with 4 replicas ($t = 1$).

Just like in MPSS [38] (see Section 2.2.4), this protocol is also leader based. Therefore, one of the servers acts as a leader and the others change the leader if they detect that it is not behaving correctly.

Since the implementation of this framework assumes secure point-to-point connections between servers, this protocol does not explicitly encrypts information transmitted during the polynomial creation (MPSS [38] does encrypt using asymmetric cryptography).

Considering the partial asynchronous system model, the polynomial creation starts when servers receive $2t + 1$ requests to create new polynomial with the same context from distinct servers. A context defines inputs for the creation of a polynomial. It is composed of an identification number, the degree of the polynomial, the fixed point it must have, the participating servers and the identification of a leader.

The generation protocol is composed of five main phases as illustrated in Figure 3.5): proposals creation, proposals collection, voting, consensus, and final point computation.

1. In the proposal creation phase, each server i creates a local polynomial F_i of degree t where $F_i(x) = y$. For each server j, i creates and sends a proposal containing the point $(j, F_i(j))$ and F_i commitments.

2. Before the voting phase begins, the leader sends to all the servers a list L containing cryptographic hashes of the first $2t + 1$ proposals it received. Since the proposal of each server contains different points, the cryptographic hash is computed using the sender's identification, and the polynomial commitments, which are equal in all servers, ensuring the proposals from the same server will have an unique cryptographic hash.

3. In the voting phase, each server j checks if the proposals the leader has selected are valid and responds to the leader listing cryptographic hashes of proposals that it considers invalid. More specifically, server j validates the proposal by checking, using commitments, if the point $(j, F_i(j))$ received from server i and the point $(x, F_i(x))$ are on the polynomial F_i.

4. In the consensus phase, servers agree on the final set of proposals S. To do so, first, the leader process the voting responses to select an S that is valid for all correct servers. Next, the leader sends

the list of voting responses using total order primitive to other servers. Those servers process this list in the same order the leader processed it to select S or, in case S is invalid, suspect the leader and servers change it, restarting the voting step. The selection of S is done in the following way: Initially, the $currentSet$ of proposals contains the $2t + 1$ proposals selected by the leader, d is 0, and $conflictList$ and $acceptList$ are empty. Then, for each voting response:

(a) If the voting response is from a replica in the $conflictList$, it is ignored.

(b) Else, if the response contains no invalid proposals against proposals in the $currentSet$, the sender is considered to be satisfied and is added to the $acceptList$.

(c) Else, the voting response has an invalid proposals still in the $currentSet$, and the replica chooses one of such proposals (in $currentSet$) deterministically (e.g., the one from the sender with lowest numbered id), removes it from this set and the responder's proposal (if it is still there), and increments d. The replica that sent the removed proposal and the sender of voting response are then both placed on the $conflictList$ and removed from the $acceptList$ (if they were on it).

When the $acceptList$ contains $2t + 1 - d$ members, this phase is completed and we have the final proposal set S.

5. In the final phase, servers compute their points of polynomial F. The final share $F(i)$ of server i is the sum of all the shares in the set S (Equation 3.1). Server i also obtains the vector of commitments for F by multiplying each commitment, at the same index, present in S (Equation 3.2).

$$F(i) = \sum_{k \in S} F_k(i) \tag{3.1}$$

$$[c_0, ..., c_t] = [\prod_{k \in S} c_{k,0}, ..., \prod_{k \in S} c_{k,t}] \tag{3.2}$$

If the leader is correct, all the replicas have selected the same S in phase 4. Otherwise, if replicas have processed all the voting responses and the $acceptList$ still does not contain $2t + 1 - d$ servers, it means that the leader is misbehaving. The parameter $d \leq t$ is an upper bound on the number of honest replicas that can end in $conflictList$, since removing an invalid proposal puts both the proposal issuer and the replica that asserted this proposal as invalid (even if this replica is honest) in $conflictList$. Therefore, $2t + 1 - d \geq t + 1$ ends up being a lower bound on the number of good replicas that have accepted the proposal set. For more details about phase 4, see the MPSS paper [38], where it was originally proposed.

3.7 Share Renewal Protocol

In long-lived services, an adversary has an unlimited amount of time to discover vulnerabilities to attack each server and obtain more than t shares and recover the confidential data. By proactively resetting servers and periodically renewing shares it is possible to avoid this threat.

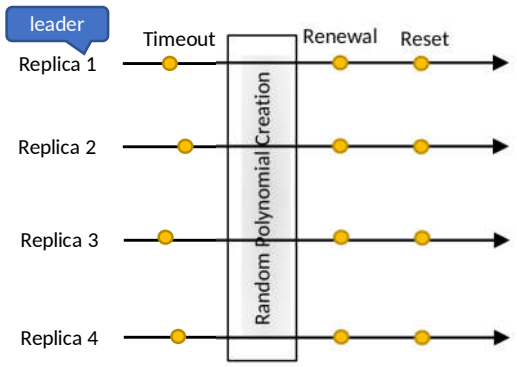

Figure 3.6: Share renewal steps in the system with 4 replicas ($t = 1$).

We use Herzberg et al. [20] scheme to renew shares of the secret without revealing the secret or the shares of other servers. More specifically, each server renews its *private data* part (assuming that each server has its part, or it can be obtained by executing the recovery protocol of Section 3.8).

When several secret shares need to be renewed, we use the same renewal polynomial Q for efficiency. Even if we use the same renewal polynomial, all the renewed polynomials of the secrets will be different because the secrets polynomials are different.

The polynomial Q is created in a distributed way using the protocol described in previous section, which guarantees that all servers will have points of the same polynomial Q, but none of them will know the polynomial.

COBRA uses a local timer to begin the share renewal process periodically. The renewal is divided into two main phases (Figure 3.6): polynomial generation and shares renewal.

The polynomial generation phase starts when there is a local timeout. Each server sends a request to generate a polynomial Q with the point $(0, 0)$. This requests will trigger the distributed polynomial generation protocol.

Servers enter into shares renewal phase when they receive their point of the polynomial Q. The servers renew shares by adding each share with its point on the polynomial Q. All the servers compute the vector of commitments of the renewed secret's polynomials by multiplying each commitment at the same index of the vectors of commitments of the previous secret's polynomial and the renewal polynomial Q. At end, each server deletes its previous share of each secret and restarts the timer.

3.8 Server Recovery Protocol

A replicated system without a recovering mechanism is not very useful, since eventually more than t replicas will fail. In SMR systems, usually, to recover the state of a replica, other servers send their state to a recovering server to restore its state. This transfer can be done collaboratively [4] to recover a big state efficiently. In SMR system that uses secret sharing, servers cannot send their state without modification, because each server stores a share of the secret. Moreover, the system can lose shares. Therefore, the combiner will not be going to be able to collect the distinct minimum number of shares.

COBRA follows the Herzberg et al. recovery scheme [20] to recover a server's shares using other servers shares, without compromising the secrecy of the secret.

The protocol to recover server i has three phases. In the first phase, the request phase, the server

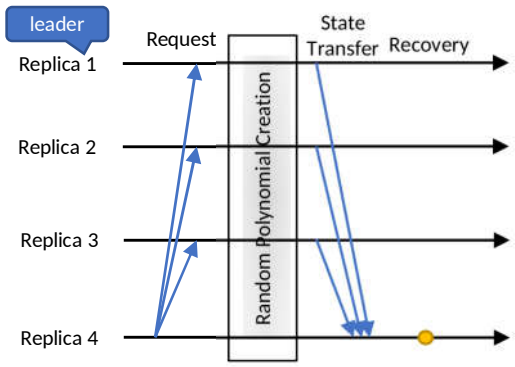

Figure 3.7: Recovery of the replica 4 in the system with 4 replicas ($t = 1$).

i requests state from other servers. In the second phase, the state transfer phase, the servers create a random recovery polynomial R for server i and send a transfer state message to i. In the third and final phase, the recovery phase, server i recovers its state using the received transfer state messages. Figure 3.7 shows the execution of this protocol to recover server 4.

In the request phase, recovering server i requests transfer state message for its *open data* and invokes the recovery protocol for its *private data* from other servers.

The state transfer phase begins when servers receive a state transfer request from i. During this phase, the servers create a random recovery polynomial R for i and create the transfer state message, which is sent to the recovering server i. The servers send new polynomial creation request to other servers with the point $(i, 0)$ to create R. This requests will trigger the distributed polynomial generation protocol (Section 3.6). When the servers receive their point on R, they create the recovering shares, which are their shares summed with their point on R. The servers create a transfer state message and send it to recovering server i.

In the recovery phase, server i rebuilds its state using the received transfer state messages. i's *open data* is extracted from the majority of replicas messages, and i uses the polynomial interpolation mechanism described in Section 3.4 to interpolate $(P + R)$ using the recovering shares to recover all its secrets shares and rebuild its *private data*. The use of this mechanism allowed us to improve the efficiency of share recovery by reducing the use of Feldman's commitments [14], as shown by the results presented in Chapter 5.

Notice that the verification of the transfer share using the commitments requires the combination of vectors of commitments of the secret's polynomial P and the recovery polynomial R since the transfer shares are of the polynomial $(P + R)$. However, the commitments of the recovered share are the same commitments of the polynomial P, stored in the *open data* part of the state.

3.9 System Reconfiguration protocol

The design of a BFT SMR system that supports long-lived services requires the implementation of a reconfiguration protocol to change the membership of participating servers and the number of faults that the system tolerate at runtime [5].

The BFT-SMaRt [1] library does have a reconfiguration protocol, supporting thus changes on the system membership and the increment/decrement of the maximum number of faults tolerated. However,

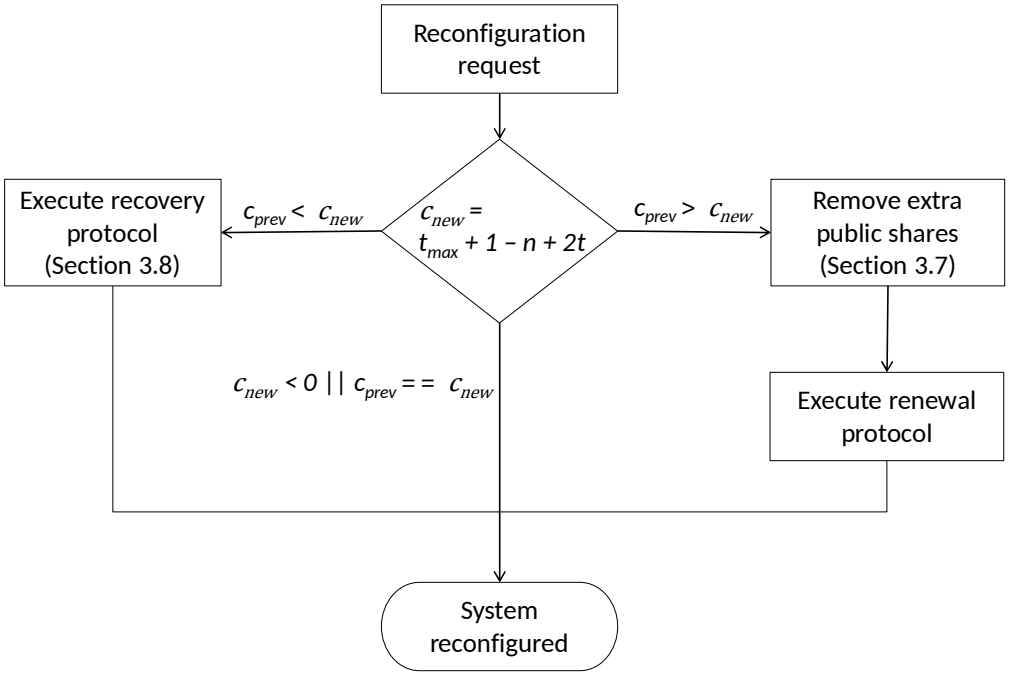

Figure 3.8: Flowchart of protocol to maintain minimum number of shares in the system after reconfiguration.

this protocol is designed to work with a system that does not have confidentiality.

When a SMR system employs secret sharing, the state of replica includes its share(s). Consequently, in this model, we have two thresholds to reconfigure, the number of faults that the SMR system tolerates and the degree of the secrets polynomials.

Adding new replicas to the system and increasing those two thresholds is not problematic, but removing replicas and decreasing the degree of the polynomials is. The addition of the new replica can be done using the recovery protocol presented in Section 3.8. Incrementing the number of faults also requires incrementing the degree of all secrets polynomials, which can be done by using the renewal protocol of the Section 3.7 using the new threshold. However, to the best of our knowledge, it is not possible to reduce the degree of a secret polynomial without revealing it. Consequently, removing a replica becomes harder because the system might not have the minimum number of shares to interpolate the correct polynomial.

In the following we sketch the solution we use in COBRA, which is similar to the one proposed in MPSS [38]. The replicas keep track of the maximum number of faults t_{max} that the system tolerated in its lifetime and, if necessary, store public shares of virtual shareholders (which have identifiers chosen from an interval defined during the system setup) to compensate the servers that left the system. This makes it possible to reduce the number of faults that SMR system tolerates and still maintains the original degree of the secret's polynomial by implementing the idea presented in the flowchart of Figure 3.8.

Since the polynomial of a secret stored in the system will have degree t_{max}, during reconfiguration a system must make sure that clients can collect $t_{max} + 1$ valid shares. Therefore, the $n - 2t$ assured responses coming from correct replicas in the current configuration must contain such valid shares. Consequently, after reconfiguration, the system must have $c_{new} = t_{max} + 1 - n + 2t$ public shares, so that shares of current correct servers $(n - 2t)$ plus c_{new} public shares (returned by any correct server) are

enough to reconstruct a polynomial of degree t_{max}.

During a reconfiguration, the system checks how many public shares currently it has (c_{prev}). If some servers left the system or the number of faults was decremented (meaning the system does not have enough shares), correct replicas recover $c_{new} - c_{prev}$ public shares for the virtual shareholders by executing the recovery protocol of Section 3.8, ensuring the system has $t_{max} + 1$ shares. Else, if a new servers were added or the number of faults was incremented, replicas remove $c_{prev} - c_{new}$ public shares of the virtual shareholders with the highest ids, since the system increased and there is no need to keep the same number of virtual shareholders to preserve $t_{max} + 1$ correct shares in the system. Additionally, replicas start the renewal protocol of Section 3.7 to change the polynomial of the secret to invalidate the public shares that some corrupted replica might not have deleted.

Notice that each call to renewal and recovery protocols uses the t_{max} value as degree of the random polynomials to be generated.

3.10 Final Remarks

In this chapter, we presented the design of COBRA protocols, for supporting the implementation of secure long-running services.

The design of COBRA integrates several SS techniques. Those techniques can be divided in two parts. The first part ensures that none of the servers knows the confidential data. The second part ensures that the private information stored in the servers stays secure even if more than t servers are compromised during the system lifetime. The former is achieved by employing the Shamir's scheme [39] augmented with Feldman's commitments [14] and the latter by employing Herzberg et al. recovery and renewal protocols [20] on top of a distributed polynomial generation protocol similar to the one used in MPSS [38]. Additionally, we sketch a solution to reconfigure the system at runtime.

Chapter 4

Implementation of Confidential BFT SMR

In this chapter we describe how we implemented COBRA in Java and how our framework was integrated to the BFT-SMaRt [1] library. This library provides all BFT SMR machinery, including reliable and authenticated communication on top of insecure communication links between replicas and between clients and replicas.

This chapter includes a description of a verifiable secret sharing library that we implemented from ground up, interfaces of COBRA that developers can use to implement services, and in the end, includes examples of two prototype applications implemented to validate COBRA.

4.1 Preliminaries

The implementation of Shamir's scheme [39] and Feldman's commitments [14] requires the existence of prime numbers q and p and a generator g of a cyclic subgroup of $\mathbb{Z}_p^*$ of order q. RFC 5114 [26] has safe values for q, p, and g that we used them to implement COBRA.

The prime q has 256-bits, and the prime p has 2048-bits. The discrete logarithm in the cyclic group $\mathbb{Z}_p^*$ is hard, which means that if the adversary does not have an unlimited computation power, then it will not be able to reverse the Feldman's commitments to obtain the coefficients of the polynomial.

4.2 Implementation Architecture

We divided the implementation of the framework into different modules, each one with a specific responsibility. Figure 4.1 shows the modules and the relationship between them.

The implementation extends BFT-SMaRt by adding secret sharing mechanisms to allow the implementation of secure long-running services. We implemented a standalone library called verifiable secret sharing with all the mechanism for dividing the confidential data and reconstructing it back. In Section 4.3, we give details about how we organized this library.

COBRA allows developers to implementca service by extending the `ConfidentialRecoverable` class on the servers and use the `ConfidentialServiceProxy` class on the client-side to communicate with the replicated service. In Section 4.4, we describe the implementation of these two classes.

The communication manager implements the agreement mechanism between servers. In Section 4.5, we describe the reason behind the implementation of this module.

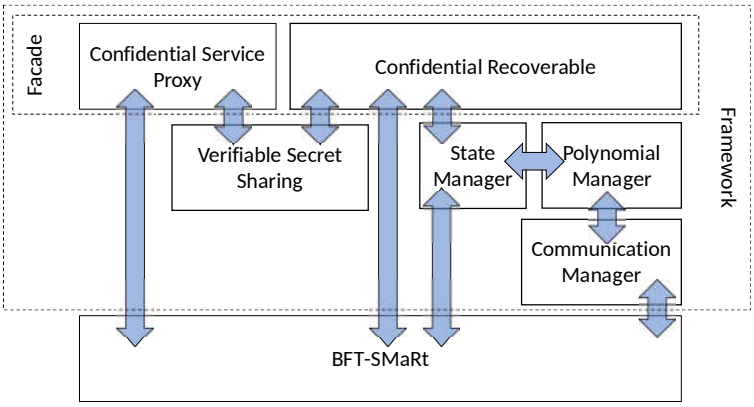

Figure 4.1: Overview of the implementation architecture.

The polynomial manager implements the distributed polynomial generation protocol. In Section 4.6, we give useful information about how we implemented this protocol.

Finally, the state manager handles share renewal and server recovery. In Section 4.7, we describe the implementation of these two protocols and give information about how we improved the implementation of the recovery protocol.

4.3 Verifiable Secret Sharing Library

We implemented a standalone verifiable secret sharing library that provides an abstraction of the SS and associated cryptographic tools. This library implements Shamir's SS [39], Feldman's commitment scheme [14], and the interpolation mechanism described in the previous chapter (Section 3.4). The implementation of those schemes are separated in different packages and are accessed through interfaces.

Since the mechanisms deal with huge numbers (up to 2048-bit), the library extensively uses the Java `java.math.BigInteger` class to represent those numbers and apply arithmetic operations (addition, subtraction, multiplication, exponentiation, and modulus) on them. All the computations are done in modulus q, except the computations related to the commitment scheme, which are done in modulus p.

The library has several classes that encapsulate information in a meaningful manner (Figure 4.2 shows the relationship between classes):

- The `Polynomial` class encapsulates a polynomial in the form $P(x) = a_t x^t + ... + a_1 x + a_0$ and has several constructors to generate this polynomial and methods to compute points of that polynomial using Horner's method [21]. Horner's method allows computing a point on the polynomial with t multiplications and t additions, instead of t exponentiations, t multiplications, and t additions.
- The `Commitments` class encapsulates vector of commitments.
- The `Share` class contains a point (shareholder id, share) of the polynomial.
- The `EncryptedShare` class represents a private `Share` and contains the encrypted share.
- The `VerifiableShare` class encapsulates the information that each server stores about a secret: the `Share`, `Commitments`, and the encrypted confidential data.

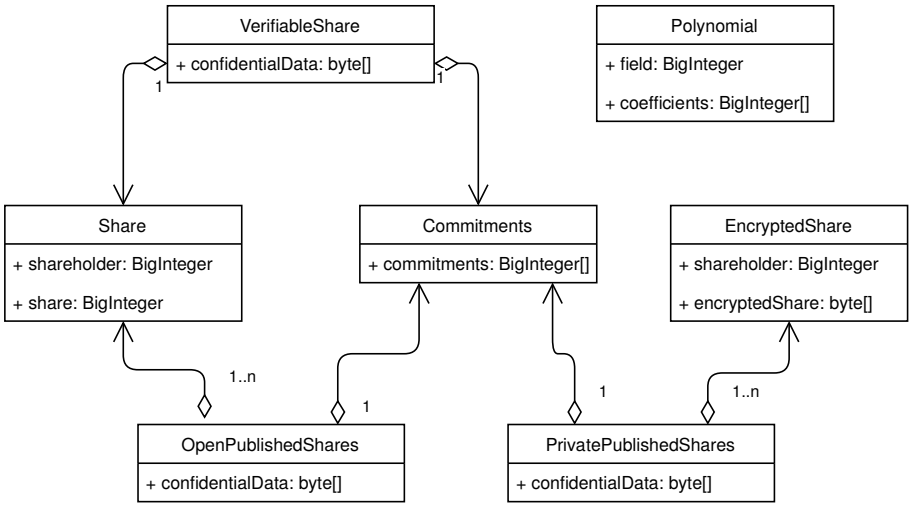

Figure 4.2: Verifiable Secret Sharing library class diagram.

- The `OpenPublishedShares` class encapsulates the `Commitments`, the encrypted confidential data, and a vector of `Share`.
- The `PrivatePublishedShares` class is a private version of the `OpenPublishedShares` that instead of a vector of `Share` has a vector of `EncryptedShare`.

We divided the implementation of the library into three main packages. Figure 4.3 shows the overview of the packages. The `vss.interpolation` package encapsulates operations related to the interpolation of the polynomials using the Lagrange method. This package exposes the interface `InterpolationStrategy`, with two methods:

- `public BigInteger interpolateAt(BigInteger x, Share[] shares)`. This method returns the value y of the point (x, y) on the polynomial interpolated using `shares`.
- `public Polynomial interpolate(Share[] shares)`. This method interpolates the polynomial P and returns it.

The `vss.commitment` package has all the computation related to the Feldman's commitment scheme [14]. The implementation of the commitment scheme exposes the interface `CommitmentScheme` with the following methods:

- `Commitments generateCommitments(Polynomial polynomial)`. This method generates the vector of commitments for the polynomial.
- `boolean checkValidity(Share share, Commitments commitments)`. This method returns true if the given share is on the polynomial and false otherwise.
- `Commitments sumCommitments(Commitments... commitments)`. This method returns the vector of commitments of the sum of the polynomials.

The `vss.secretsharing` package uses the above two packages to share confidential data and to reconstruct it back. The package implements the data protection scheme described in the previous chapter. To use this scheme, the package exposes the interface `VSSFacade` with the following methods:

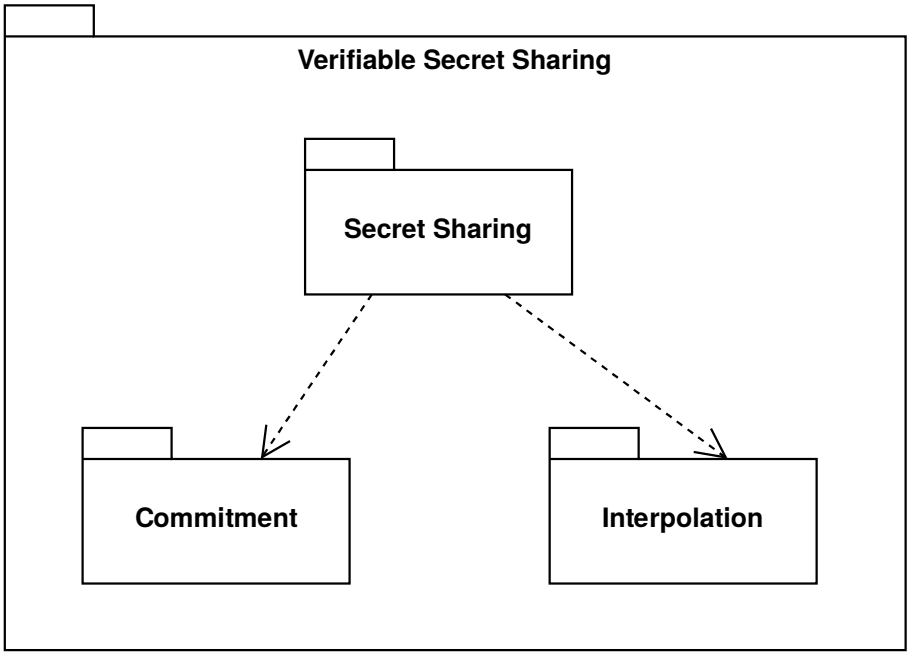

Figure 4.3: Verifiable Secret Sharing library package diagram.

- `public PrivatePublishedShares share(int threshold, byte[] data, Map<BigInteger, Key> encryptionKeys)`. To generate random key, we use the `java.security.SecureRandom` class to generate a random array of 256 bits such that the `BigInteger` created with this array corresponds to a positive number. The data is encrypted using AES-256 and each share is encrypted using keys provided in the `encryptionKeys` map.

- `public byte[] combine(OpenPublishedShares openShares)`. To create a correct polynomial, this method employs the mechanism described in Section 3.4.

- `public VerifiableShare extractShare(PrivatePublishedShares privateShares, BigInteger shareholder, Key decryptionKey)`. Each replica uses this method to extract its share.

The implementation of this library was tested using the *JUnit* framework. We test each scheme with differents examples of random generated data.

4.4 Framework Facade

The framework exposes two classes to implement the service. The `ConfidentialServiceProxy` used on the client-side to communicate with the replicated service, which is implemented, by each server, by extending the abstract class `ConfidentialRecoverable`. Appendix A lists the Java code of both of these classes.

The `ConfidentialServiceProxy` uses the `ServiceProxy` class of BFT-SMaRt to communicate with the replicated system and the `VerifiableSecretSharing` library to perform all the operations related to the secret sharing mechanism.

The client uses the instance of `ConfidentialServiceProxy` to send requests to the service running on the replicate system. The class has two methods:

- `public Response invokeOrdered(byte[] plainData, byte[]... confidentialData)`. The requests sent using this method are ordered by the replicas before delivering them to the service.

- `public Response invokeUnordered(byte[] plainData, byte[]... confidentialData)`. The requests sent using this method are delivered to the service without executing the consensus.

Both methods accept an array of bytes of plain data, which all the replicas in the system will receive without modification. Moreover they accept zero or more arrays of bytes of confidential data, which the framework divides into shares for each replica.

To reconstruct the confidential data using responses, the framework calls the method `VSSFacade::combine`, but first instantiates the `OpenPublishedShares` using replicas responses. The selected vector of commitments is the vector of commitments sent by the majority of replicas and the same goes for the encrypted confidential data. The implementation also takes into account public shares that each replica sends.

The abstract class `ConfidentialRecoverable` receives request from BFT-SMaRt and uses the `VerifiableSecretSharing` library to extract its share from the request. Before delivering its share to the service, this class encapsulates the share in a `ConfidentialData` object containing the server's share, all the public shares (created during system reconfiguration) of the same secret, and the public information (commitments and encrypted data). Additionally, this class handles logging and periodically makes checkpoints to trim the log. To extend this abstract class, a service has to implement the following abstract methods:

- `public abstract ConfidentialMessage appExecuteOrdered(byte[] plainData, ConfidentialData[] shares, MessageContext msgCtx)`. Called when an ordered request is delivered.

- `public abstract ConfidentialMessage appExecuteUnordered(byte[] plainData, ConfidentialData[] shares, MessageContext msgCtx)`. Called when an unordered request is delivered.

- `public abstract ConfidentialSnapshot getConfidentialSnapshot()`. Called when the library needs a snapshot of the service state. The `ConfidentialSnapshot` contains an array of bytes of public data and a list of shares.

- `public abstract void installConfidentialSnapshot(ConfidentialSnapshot snapshot)`. Called to reset the service state to the state contained in a snapshot.

4.5 Communication Between Servers

BFT-SMaRt does not have an explicit functionality to send ordered messages between servers. However, the implementation of our distributed polynomial generation protocol requires the leader to send a list of voting responses using a total order primitive.

We implemented this functionality using the mechanism that the library has to forward messages to the leader when the first timeout happens during the consensus [5]. The implementation did not require to modify the library itself. Instead, we implemented a module that allows the distributed polynomial generation protocol module to send messages between servers with and without ordering. This module encapsulates the messages as they were forwarded and sends it to all the servers.

It is necessary to subscribe to receive messages that come through this module. Each message sent using it has a field called type that is assigned by the sender, and the receiver who subscribed for this type of message will receive it through a listener object.

4.6 Distributed Polynomial Generation Protocol

We implemented the distributed polynomial generation protocol as a primitive that servers can use to generate their points on the same global polynomial. The execution of this protocol runs in a separate thread. Consequently, while the polynomials are being created, servers continue to execute clients requests.

The servers create a context of the polynomial to start its creation. The leader for each polynomial is the same as the current leader in BFT-SMaRt. Moreover, each polynomial creation has a field called reason that specifies for what purpose this polynomial is being created. This field, for now, has one of two values: `RECOVERY` or `RESHARING`.

All the servers in the system participate in generating the polynomial. Servers ignore the point if they do not need it, which is the case when recovering a server.

4.7 Share Renewal and Recovery

The renewal and recovery protocols modify the state of the service. The former modifies shares and applies the state back to the service, while the latter modifies the state and sends it to the server being recovered. In both cases, the state being modified is retrieved from the log and not from the service.

During renewal, the modified state of the log is applied back to log and the service state is resetted according to the new log.

During the recovery, the log state is separated into public state and a list of `Share` objects. The public state contains the public data and information about how to join the `Share`. The goal of the separation is to begin the transmission of the public state when the recovering request arrives, while the recovery polynomial is generated to compute the transfer of shares.

4.8 Examples of Applications

To validate our framework, we implemented two prototype applications using COBRA: a key-value store and a dependable tuple space.

4.8.1 Private Key-Value Store

This application is a simple implementation of Java's `java.util.Map` interface. However, each server stores hashes of keys and a share of the value. Thus if simultaneously at most f servers are compromised, the values are not revealed. Nevertheless, an adversary can learn keys even if he/she attacks one server by using a hash dictionary.

On the server-side, the framework's class `ConfidentialRecoverable` is extended to implement the `Map` interface. On the client-side, the framework also implements the `Map` interface, but instead of storing the data it sends to the server using `ConfidentialServiceProxy`.

In our current prototype we only implement the following operations: `put`, `get`, `remove`, `values`. Furthermore, we provide a console client to use this service.

4.8.2 DepSpace

DepSpace [6] is a dependable tuple space implemented on top of BFT-SMaRt. Although the system supports confidentiality, it is only safe if in an entire lifetime of the service an adversary only compromises at most f servers. We integrate our framework in DepSpace (replacing its confidentiality layer) to make it secure for the entire lifetime of the service.

On the client-side stack of the DepSpace, we removed the `ClientConfidentialityLayer` and modified the `ClientReplicationLayer` to use `ConfidentialServiceProxy` instead of `DepSpaceServiceProxy`. On the server-side stack, we removed the `ServerConfidentialityLayer` and modified the `DepSpaceReplica` to extend `ConfidentialRecoverable` instead of `DefaultRecoverable`.

4.9 Final Remarks

This chapter explains how we implemented COBRA by extending BFT-SMaRt. The implementation required the design and implementation of a library with SS from ground up using the `BigInteger` class to represent big numbers.

COBRA exposes two classes to allow implementation of services. To validate our framework, we implemented two applications. The first one is a private key-value store where each server stores a share of the value and the second is a dependable tuple space.

Chapter 5

Evaluation

In this chapter, we evaluate the overhead of our verifiable secret sharing library and the integration of COBRA in BFT-SMaRt. With this evaluation, we want answers the following questions:

- How many additional bytes are sent from a client to servers in our system?
- How many additional bytes each server stores for each confidential data?
- What is the computational overhead of employing secret sharing in a SMR system?
- How good is the performance of out distributed polynomial generation protocol?
- What is the impact of renewing and recovering a large number of shares?
- What is the impact of integrating secret sharing layer in BFT-SMaRt?

All tests were executed in machines with the following characteristics: Dell PowerEdge R410 with two quad-core Intel Xeon E5520 (2.27 GHz) CPUs; two hardware threads per core; 32 GB of RAM; Ubuntu 10.04 operating system with Oracle Java SE version 1.8.0_211.

5.1 Transmission and Storage Cost of Secret Sharing

COBRA's data protection scheme sends and stores, at each server, the encrypted confidential data. To store confidential data into servers, the client has to send the shares of each server and the vector of commitments, besides the encrypted data. Each server has to store its share, the vector of commitments, and the encrypted data.

To measure the impact of secret sharing, we calculated the cost of transmission and storage of shares and the vector of commitments based on the public parameters. There costs are presented in Tables 5.1 and 5.2. Recalling that the public parameters q has 32 bytes and p of 256 bytes.

Table 5.1 shows that the cost of transmission increases linearly in function of the number of faults tolerated because there are more servers in the system and the degree of secret's polynomial is same as the number of faults. However, the vector of commitments corresponds to a considerable percentage of this cost. Similarly, Table 5.2 shows that the storage cost also increases linearly in function of the number of faults. More specifically, it increases 256 bytes for each increment in t since only the number of commitments increases.

t	1	2	3	10
Total	640	992	1344	3808
Shares	128	224	320	992
Commitments	512	768	1024	2816

Table 5.1: The additional cost in bytes of the request to write confidential data from client to the servers in function of number of tolerated faults ($n = 3t + 1$)

t	1	2	3	10
Total	544	800	1056	2848
Share	32	32	32	32
Commitments	512	768	1024	2816

Table 5.2: The additional cost in bytes of storage space occupied by confidential data at each replica in function of number of tolerated faults ($n = 3t + 1$).

5.2 Computation cost of Secret Sharing

In this section, we present the secret sharing computation costs. Each value presented is an average of 25 executions of the same computation in a single thread. The standard deviation is under 5%.

5.2.1 Commitments

The commitments of a polynomial allows checking if a point is on the polynomial or not. Table 5.3 shows the time required, in milliseconds, for the creation of vector of commitments and for using it to verify $t + 1$ shares. The results show that the client has a notable cost when writing and reading confidential data. The results also indicate that it is not practical to use just Feldman's commitment scheme [14] to verify shares of a significant number of secrets, justifying our choice for an hybrid scheme, evaluated in next section.

t	1	2	3	10
Generation	2.739	4.204	5.389	14.406
Verification	2.848	4.450	5.976	30.301

Table 5.3: The cost in milliseconds of Feldman's commitment scheme [14] in function of number of tolerated faults ($n = 3t + 1$).

5.2.2 Polynomial Interpolation

In this section we compare the impact of Feldman's commitment scheme [14] with OPI scheme (Section 3.4). Table 5.4 shows that, independent of faulty points, commitments are used to verify at least $t+1$ points. However, in OPI, commitments are used only when the detection scheme [18] finds invalid points and at most to identify t corrupted servers, which goes to a $blacklist$. Therefore when there are no invalid points, the cost is minimal because the polynomial interpolation goes through a fast path (Figure 3.2).

During multiple polynomials interpolation (Table 5.5), the cost of OPI is significantly lower compared to the commitment scheme, because during the first t polynomials interpolation corrupted servers are identified and added to the $blacklist$ using commitments. Therefore, the remaining interpolations goes through the fast path since the points of servers in the $blacklist$ are not used.

Nº of invalid shares	0	1	2	3
Commitment scheme	7.720	8.015	8.308	8.967
OPI	0.192	8.191	8.465	8.820

Table 5.4: The cost in milliseconds of 1 correct polynomial interpolation in a system that tolerates at most 3 faults ($n = 10, t = 3$).

Nº of invalid shares	0	1	2	3
Commitment scheme	122.433	126.380	131.093	136.128
OPI	2.040	10.126	10.344	10.029

Table 5.5: The cost in milliseconds of 16 correct polynomial interpolation in a system that tolerates at most 3 faults ($n = 10, t = 3$).

5.2.3 Share and Combine

Table 5.6 presents the overall computation latency of our data protection scheme. To write 16 confidential data entries, the creation of commitments has high cost compared to the shares generation. However, to read confidential data, the overall total cost is much lower than the total cost of writing, due to our OPI scheme.

Nº of invalid shares		0	1	2	3
Write	Shares generation	0.872	0.874	0.876	0.868
	Commitments Generation	106.484	106.543	106.714	106.706
Read	Share verification	0.000	7.775	8.062	8.371
	Confidential data reconstruction	2.289	2.581	2.366	1.901

Table 5.6: The cost in milliseconds of writting and reading 16 confidential data entries of 1 kB in a system that tolerates at most 3 faults ($n = 10, t = 3$).

5.3 Polynomial Generation Cost

The distributed polynomial generation protocol is used to create renewal and recovery polynomials. In Figure 5.1 we present a comparison between the polynomial generation cost in MPSS and in our protocol during a renewal (which is expected to be executed periodically) for system with $n = 4$ servers ($t = 1$). In our implementation of MPSS the protocol was changed to use symmetric cryptography. Since we wanted to measure the computation cost, the latency does not include the communication between replicas.

As can be observed in the figure, the MPSS polynomial generation took approximately 37 milliseconds, and our protocol took approximately 9.5 milliseconds. This is due to the fact that MPSS generates $1 + n$ polynomials (1 renewal and n recovery) and our protocol only requires to generate 1 renewal polynomial (because we separated renewal from recovery). As a result, in our protocol, the proposals phase takes about 2.8 milliseconds and the voting phase about 6.5 milliseconds since commitments are used to verify points of a single polynomial. In MPSS, the proposals phase took about 15.5 milliseconds and the voting phase about 21 milliseconds because commitments are used to verify points of $1 + n$ polynomials.

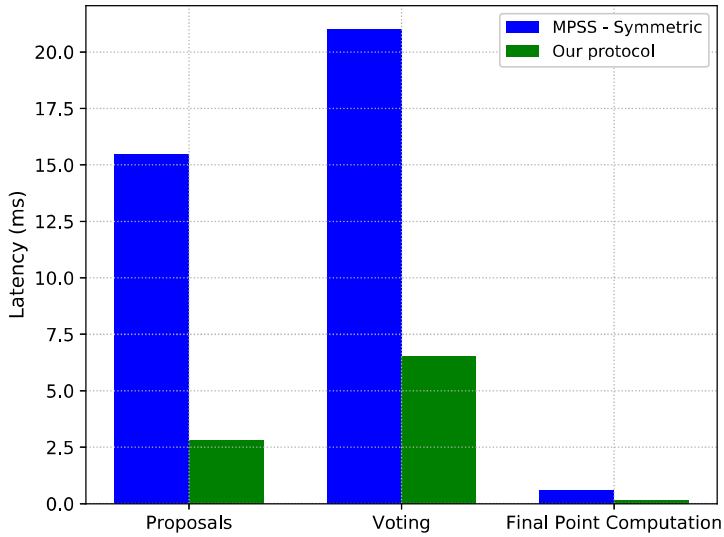

Figure 5.1: Comparison of polynomial generation cost between MPSS [38] polynomial generation part and our polynomial generation protocol.

5.4 Renewal and Recovery Cost

The renewal and recovery protocols are used to, respectively, renew and recovery the shares contained in the *private data* of servers state. Figure 5.2 presents the computation cost of these protocols. As mentioned in the previous section, MPSS [38] combines renewal and recovery into one single protocol and uses Feldman's commitment scheme [14]. Consequently, frequently renewing a significant number of shares is not practical because each replica needs to verify the shares received from other replicas using commitments, which has a high cost.

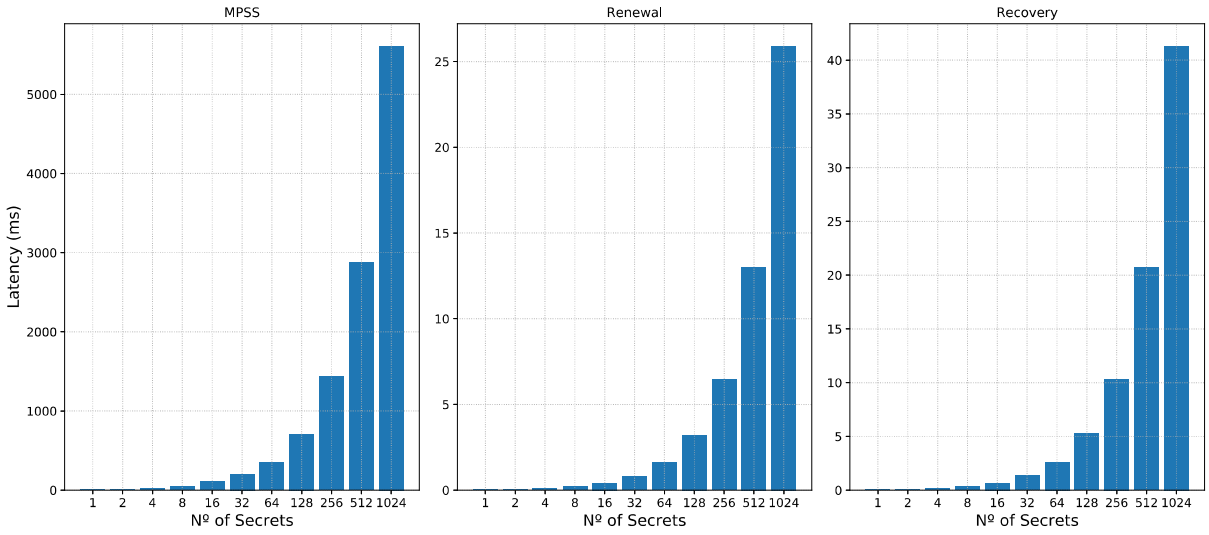

Figure 5.2: Execution cost of renewal and recovery protocols. Executed as one step (MPSS [38]) and executed separately.

However, the shares used during renewal are of the replica itself, so there is no need to verify it.

Therefore, it is possible to renew 1024 shares in about 26 milliseconds in COBRA, while in MPSS that takes more than 5 seconds. Moreover, by using the OPI scheme, COBRA can recover 1024 shares in about 41 milliseconds in the absence of faults (using fast path - Figure 3.2), which is 100x faster than MPSS. The results presented in Section 5.2.2 show that even if there are invalid shares, the cost of using commitments during interpolation is limited, which means that our recovery protocol will still be significantly faster than MPSS.

5.5 COBRA End-to-End Performance

In this section we present the impact of using COBRA to implement secure long-running services. We configured the system with four replicas (one on each physical machine), tolerating one Byzantine fault, and a number of clients spread on other two machines. All these machines follow the specification described before.

We use YCSB [2] to generate load. YCSB is a framework that allows generating configurable workloads to evaluate the performance of a key-value store service. Our evaluation is based on the private key-value store described in Section 4.8.1.

5.5.1 Read and Write Data

Figures 5.3 and 5.4 shows the impact of integrating confidentiality to BFT SMR by executing puts and gets in the private key-value store, respectively.

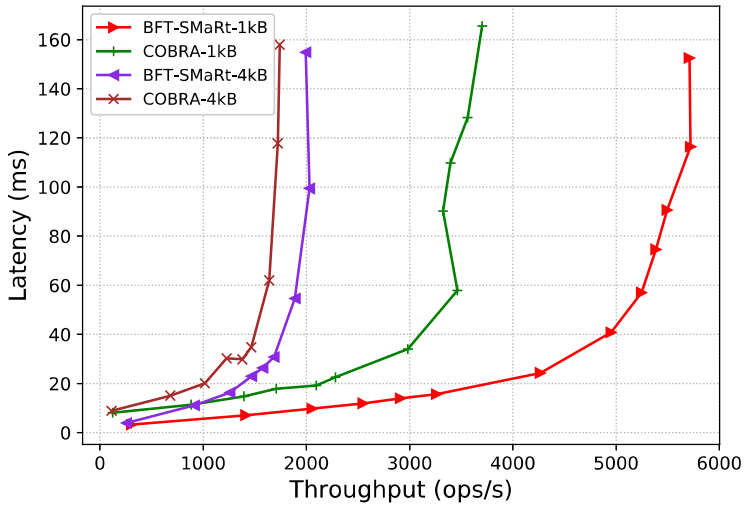

Figure 5.3: Throughput vs. Latency graph of 100% write workload in a system with $n = 4$ replicas ($t = 1$).

COBRA has a significant more impact in the system with 100% write workload (Figure 5.3) when compared with the same system with 100% read workload (Figure 5.4). This difference is due to the commitment scheme. Readings are faster because of the OPI scheme when all the servers are correct, since there is no need to verify shares using commitments.

As expected, in both cases, the throughput of system without confidentiality is higher than the throughput of the system with confidentiality. However, the difference decreases when the size of the

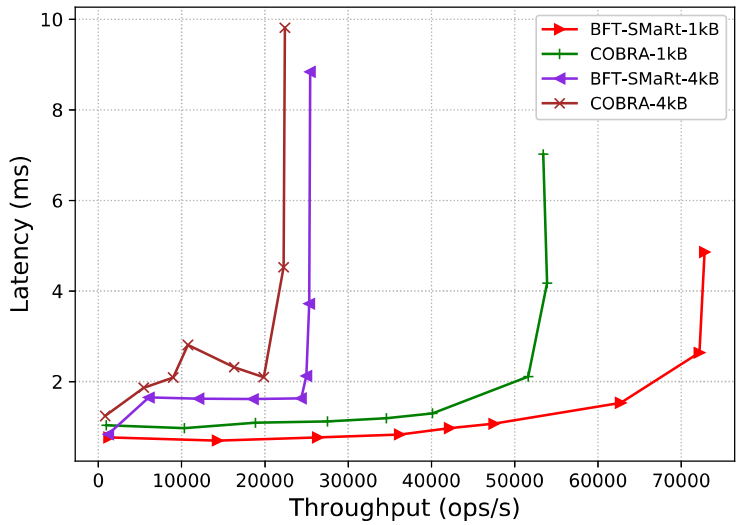

Figure 5.4: Throughput vs. Latency graph of 100% read workload in a system with $n = 4$ replicas ($t = 1$).

data increases, since data transfer becomes a bottleneck. Specifically, BFT-SMaRt without COBRA has approximately 35% higher throughput than with it when writing 1kB of data. However, when writing 4kB of data, the throughput is only 13% higher. Similarly, when reading 1kB of data, BFT-SMaRt has approximiately 26% higher throughput than COBRA, but the difference decreases to 13% for data sizes of 4kB.

Because of the our OPI scheme, 100% read workload has approximately 93% higher throughput than 100% write workload with the same amount of data. However, this result is not definitive since during the experiments we noticed that the more recent BFT-SMaRt version used in this thesis does not have same write throughput of previous versions. We expect to clarify this in the next months and redo these experiments.

5.5.2 State Recovery

The recovery of a server requires other replicas to generate recovery shares for the *private data* of such server and send it with the corresponding *open data*. Figure 5.5 presents the impact of recovering about 200000 data entries (key-value pairs) of 1kB in COBRA, which corresponds to a state of 1 GB (*open data* plus BFT-SMaRt data) and about 200000 shares. To compare with a system without confidentiality, Figure 5.6 presents the impact of recovering about 200000 data entries (key-value pairs) of 1kB in a BFT-SMaRt-based key-value store.

As can be observed, COBRA recovery is about 2× slower when compared to BFT-SMaRt. In both cases, recovering a server impacts the whole system. However, in COBRA this impact is significantly higher, due to the computation of the recovering shares. Moreover, the state transfer is a bottleneck more visible in COBRA since it has to transfer, in addition to the confidential data, the information related to the secret sharing. Due to the confidentiality, installing state in the service takes about 31 seconds in COBRA, and 16 seconds in BFT-SMaRt. This difference comes mostly the fact that installing state requires first to rebuild recovering server's *private data* (i.e., its shares), an operation that lasts more than

10 seconds.

In the future, we want compute recovering shares in a separate thread and making only one selected replica send the *open data* while the others will only send cryptographic hashes of it (as it is done in BFT-SMaRt). We expect this to significantly decreases both the impact and the time to recover a replica in COBRA.

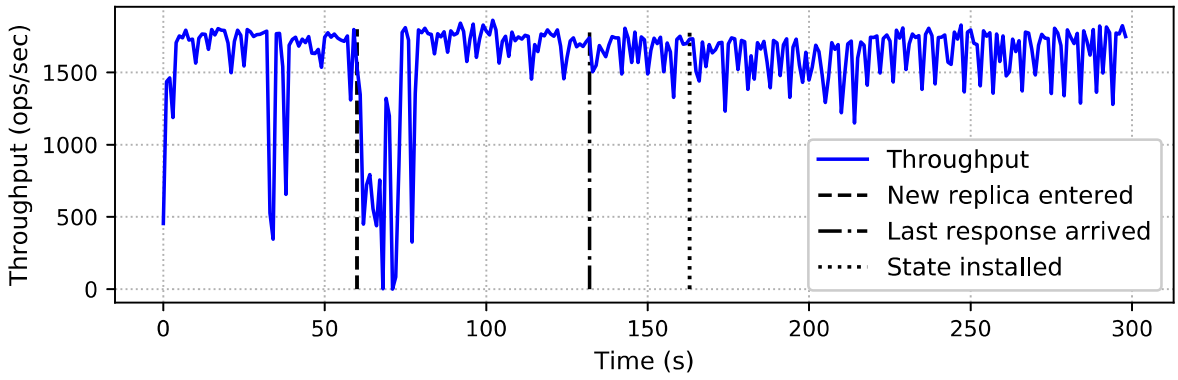

Figure 5.5: Impact of recovery protocol in COBRA in the system with $n = 4$ replicas ($t = 1$).

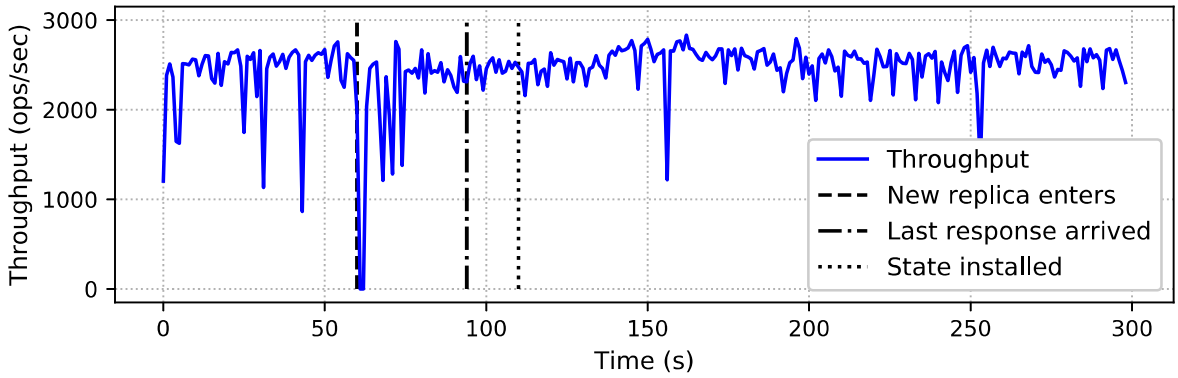

Figure 5.6: Impact of recovery protocol in BFT-SMaRt in the system with $n = 4$ replicas ($t = 1$).

5.6 Final Remarks

In this chapter we presented a preliminary evaluation of COBRA. As expected, the integration of a secret sharing layer has negative impact compared to a BFT SMR system that does not support confidentiality. However, we significantly improved reading throughput by proposing the OPI scheme, which combines previously known commitment [14] and detection [18] schemes. Also, OPI allowed us to efficiently recover a large number of shares during state recovery. Moreover, the results show that COBRA significantly improves the distributed polynomial generation by modifying how the proposals are generated and transmitted.

Chapter 6

Conclusion and Future Work

In this master dissertation we presented the design and implementation of a confidentiality framework for BFT systems called COBRA. COBRA allows implementing fault- and intrusion-tolerant services using state machine replication. Confidentiality protection is achieved by employing different secret sharing schemes to store data into servers. Additionally, COBRA protects services against a mobile adversary by enabling periodic renewals of the shares stored in the service.

Earlier in this work we found, from the experiments we did, that the use of commitments would have a significant cost in terms of performance. However, these commitments allow the identification of invalid shares in systems with a higher number of faults (compared with another competing scheme [18]). Hence, we proposed an hybrid OPI (Optimistic Polynomial Interpolation) scheme to mitigate the cost of using commitments. We made COBRA scalable to a large number of shares by integrating this OPI scheme into its design. As a result, for example, our replica recovery protocol is $100\times$ faster than a previous protocol (MPSS) when recovering 1024 shares.

Finally, we want to mention that during a couple of months in this project we tried, without success, to find a way to compress recovering shares transmitted during the recovery of a server. The idea was to mitigate the cost of transmission and verification of recovering shares by encoding it into a polynomial. Luckily, we found a way to attenuate the cost the recovery protocol by using our hybrid OPI scheme.

6.1 Future Work

During our preliminary experimental evaluation, we identified some problems in our prototype. More importantly, we identified an issue with BFT-SMaRt write throughput, that prevented us from finding out the real impact of the integration of secret sharing to BFT SMR. Hence, in future work we want to investigate and amend such performance issues, and reevaluate COBRA to extract correct numbers.

Similarly, we want also to conduct more experiments to measure the impact of renewal and recovery in the system. More specifically, the cost of generating polynomials and comparing recovery with BFT-SMaRt for different state sizes and replica groups. In the same way, we want to properly evaluate the share renew and group reconfiguration protocols.

The integration of the OPI scheme allowed reducing the verification of shares using Feldman's commitments [14]. However, COBRA still use such commitments, and pays for the significant transmission and storage overhead they bring. To mitigate such (passive) cost, we want to investigate Kate at el. commitment scheme [22], which has a constant cost of transmission and storage.

We use AES to encrypt confidential data. Thus, a service can not perform computation on the encrypted data. Hence, we want to investigate how symmetric homomorphic encryption schemes can be used in our system. Additionally, if the chosen scheme has a property that allows detecting the use of an invalid key during decryption, we can eliminate the need for Harn and Lin detection scheme [18]. Therefore, in the absence of faults, there is no need to wait for an additional share ($t + 2$ instead of $t + 1$ shares) during confidential data reconstruction.

Appendix A

COBRA's Facade Java Code

A.1 The `ConfidentialServiceProxy` class

```
package confidential.client;

import bftsmart.tom.ServiceProxy;
import bftsmart.tom.util.Extractor;
import confidential.ExtractedResponse;
import confidential.MessageType;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import vss.facade.SecretSharingException;
import vss.secretsharing.OpenPublishedShares;
import vss.secretsharing.PrivatePublishedShares;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.util.Comparator;

public class ConfidentialServiceProxy {
    private final Logger logger = LoggerFactory.getLogger("confidential");

    private final ServiceProxy service;
    private final ClientConfidentialityScheme confidentialityScheme;

    public ConfidentialServiceProxy(int clientId) throws SecretSharingException {
        Extractor extractor = new ConfidentialExtractor();
        Comparator<byte[]> comparator = new ConfidentialComparator();

        this.service = new ServiceProxy(clientId, null, comparator, extractor, null)
    ;
        this.confidentialityScheme = new ClientConfidentialityScheme(service.
    getViewManager().getCurrentView());
    }

    public Response invokeOrdered(byte[] plainData, byte[]... confidentialData)
    throws SecretSharingException {
        byte[] request = composeRequest(plainData, confidentialData);
        if (request == null)
            return null;

        byte[] response = service.invokeOrdered(request);

        return composeResponse(response);
    }
```

```
43      public Response invokeUnordered(byte[] plainData, byte[]... confidentialData)
        throws SecretSharingException {
            byte[] request = composeRequest(plainData, confidentialData);
45          if (request == null)
                return null;
47
            byte[] response = service.invokeUnordered(request);
49
            return composeResponse(response);
51      }

53      public void close() {
            service.close();
55      }

57      private Response composeResponse(byte[] response) throws SecretSharingException
        {
            if (response == null)
59              return null;
            ExtractedResponse extractedResponse = ExtractedResponse.deserialize(response
        );
61          if (extractedResponse == null)
                return null;
63          OpenPublishedShares[] openShares = extractedResponse.getOpenShares();
            byte[][] confidentialData = openShares != null ? new byte[openShares.length
        ][] : null;
65          if (openShares != null) {
                for (int i = 0; i < openShares.length; i++) {
67                  confidentialData[i] = confidentialityScheme.combine(openShares[i]);
                }
69          }
            return new Response(extractedResponse.getPlainData(), confidentialData);
71      }

73      private byte[] composeRequest(byte[] plainData, byte[]... confidentialData)
        throws SecretSharingException {
            try (ByteArrayOutputStream bos = new ByteArrayOutputStream();
75               ObjectOutput out = new ObjectOutputStream(bos)) {

77              out.write((byte)MessageType.CLIENT.ordinal());

79              out.writeInt(plainData == null ? -1 : plainData.length);
                if (plainData != null)
81                  out.write(plainData);

83              out.writeInt(confidentialData == null ? -1 : confidentialData.length);
                if (confidentialData != null) {
85                  PrivatePublishedShares privateShares;
                    for (byte[] secret : confidentialData) {
87                      privateShares = confidentialityScheme.share(secret);
                        privateShares.writeExternal(out);
89                  }
                }
91
                out.flush();
93              bos.flush();
                return bos.toByteArray();
95          } catch (IOException e) {
                logger.error("Occurred while composing request", e);
97              return null;
            }
99      }
```

```
}
```

A.2 The `ConfidentialRecoverable` class

```
package confidential.server;

import bftsmart.reconfiguration.ReconfigureRequest;
import bftsmart.reconfiguration.ServerViewController;
import bftsmart.statemanagement.ApplicationState;
import bftsmart.statemanagement.StateManager;
import bftsmart.tom.MessageContext;
import bftsmart.tom.ReplicaContext;
import bftsmart.tom.server.Recoverable;
import bftsmart.tom.server.SingleExecutable;
import bftsmart.tom.server.defaultservices.CommandsInfo;
import bftsmart.tom.server.defaultservices.DefaultApplicationState;
import bftsmart.tom.util.TOMUtil;
import confidential.ConfidentialData;
import confidential.ConfidentialMessage;
import confidential.MessageType;
import confidential.interServersCommunication.InterServersCommunication;
import confidential.polynomial.DistributedPolynomial;
import confidential.statemanagement.ConfidentialSnapshot;
import confidential.statemanagement.ConfidentialStateLog;
import confidential.statemanagement.ConfidentialStateManager;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import vss.facade.SecretSharingException;
import vss.secretsharing.PrivatePublishedShares;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.locks.ReentrantLock;

public abstract class ConfidentialRecoverable implements SingleExecutable,
    Recoverable {
    private Logger logger = LoggerFactory.getLogger("confidential");
    private ServerConfidentialityScheme confidentialityScheme;
    private final int processId;
    private ReplicaContext replicaContext;
    private ConfidentialStateLog log;
    private ReentrantLock stateLock;
    private ReentrantLock logLock;
    private ConfidentialStateManager stateManager;
    private InterServersCommunication interServersCommunication;
    private int checkpointPeriod;
    private List<byte[]> commands;
    private List<MessageContext> msgContexts;
    private int currentF;

    public ConfidentialRecoverable(int processId) {
        this.processId = processId;
        this.logLock = new ReentrantLock();
        this.commands = new ArrayList<>();
        this.msgContexts = new ArrayList<>();
    }
```

```
58      @Override
        public void setReplicaContext(ReplicaContext replicaContext) {
60          logger.debug("setting replica context");
            this.currentF = replicaContext.getSVController().getCurrentViewF();
62          this.replicaContext = replicaContext;
            this.stateLock = new ReentrantLock();
64          interServersCommunication = new InterServersCommunication(
                    replicaContext.getServerCommunicationSystem(), replicaContext.
    getSVController());
66          checkpointPeriod = replicaContext.getStaticConfiguration().
    getCheckpointPeriod();
            try {
68              this.confidentialityScheme = new ServerConfidentialityScheme(processId,
    replicaContext.getCurrentView());
                DistributedPolynomial distributedPolynomial = new DistributedPolynomial(
    processId, interServersCommunication,
70                      confidentialityScheme.getCommitmentScheme(),
    confidentialityScheme.getField());
                new Thread(distributedPolynomial, "Distributed polynomial").start();
72              stateManager.setDistributedPolynomial(distributedPolynomial);
                stateManager.setInterpolationStrategy(confidentialityScheme.
    getInterpolationStrategy());
74              stateManager.setCommitmentScheme(confidentialityScheme.
    getCommitmentScheme());
                log = getLog();
76              stateManager.askCurrentConsensusId();
            } catch (SecretSharingException e) {
78              logger.error("Failed to initialize ServerConfidentialityScheme", e);
            }
80      }

82      private ConfidentialStateLog getLog() {
            if (log == null)
84              log = initLog();
            return log;
86      }

88      private ConfidentialStateLog initLog() {
            if (!replicaContext.getStaticConfiguration().isToLog())
90              return null;
            ConfidentialSnapshot snapshot = getConfidentialSnapshot();
92          byte[] state = snapshot.serialize();
            if (replicaContext.getStaticConfiguration().logToDisk()) {
94              logger.error("Log to disk not implemented");
                return null;
96          }
            logger.info("Logging to memory");
98          return new ConfidentialStateLog(processId, checkpointPeriod, state, TOMUtil.
    computeHash(state));
        }
100
        @Override
102     public ApplicationState getState(int cid, boolean sendState) {
            logLock.lock();
104         logger.debug("Getting state until CID {}", cid);
            ApplicationState state = (cid > -1 ? getLog().getApplicationState(cid,
    sendState)
106                 : new DefaultApplicationState());
            if (state == null ||
108                 (replicaContext.getStaticConfiguration().isBFT()
                            && state.getCertifiedDecision(replicaContext.getSVController
    ()) == null))
```

```
110             state = new DefaultApplicationState();
            logLock.unlock();
112         return state;
        }
114
        @Override
116     public int setState(ApplicationState recvState) {
            int lastCID = -1;
118         if (recvState instanceof DefaultApplicationState) {
                DefaultApplicationState state = (DefaultApplicationState)recvState;
120             logger.info("I'm going to update myself from CID {} to CID {}",
                        state.getLastCheckpointCID(), state.getLastCID());
122
                stateLock.lock();
124             logLock.lock();
                log.update(state);
126
                int lastCheckpointCID = log.getLastCheckpointCID();
128             lastCID = log.getLastCID();
130
                if (state.getSerializedState() != null) {
132                 logger.info("Installing snapshot up to CID {}", lastCheckpointCID);
                    ConfidentialSnapshot snapshot = ConfidentialSnapshot.deserialize(
    state.getSerializedState());
134                 installConfidentialSnapshot(snapshot);
                }
136
                for (int cid = lastCheckpointCID + 1; cid <= lastCID; cid++) {
138                 try {
                        logger.debug("Processing and verifying batched requests for CID
    {}", cid);
140                     CommandsInfo cmdInfo = log.getMessageBatch(cid);
                        if (cmdInfo == null) {
142                         logger.warn("Consensus {} is null", cid);
                            continue;
144                     }
                        byte[][] commands = cmdInfo.commands;
146                     MessageContext[] msgCtx = cmdInfo.msgCtx;

148                     if (commands == null || msgCtx == null || msgCtx[0].isNoOp())
                            continue;
150
                        for (int i = 0; i < commands.length; i++) {
152                         Request request = Request.deserialize(commands[i]);
                            if (request == null) {
154                             logger.warn("Request is null");
                                continue;
156                         }
                            if (request.getType() == MessageType.APPLICATION) {
158                             logger.debug("Ignoring application request");
                                continue;
160                         }
                            appExecuteOrdered(request.getPlainData(), request.getShares
    (), msgCtx[i]);
162                     }
                    } catch (Exception e) {
164                     logger.error("Failed to process and verify batched requests for
    CID {}", cid, e);
                        if (e instanceof ArrayIndexOutOfBoundsException) {
166                         logger.info("Last checkpoint CID: {}", lastCheckpointCID);
                            logger.info("Last CID: {}", lastCID);
```

```
                    logger.info("Number of messages expected to be in the batch: {}", (log.getLastCID() - log.getLastCheckpointCID() + 1));
                    logger.info("Number of messages in the batch: {}", log.getMessageBatches().length);
                }
            }
        }
        logLock.unlock();
        stateLock.unlock();
    }
    return lastCID;
}

@Override
public StateManager getStateManager() {
    if (stateManager == null)
        stateManager = new ConfidentialStateManager();
    return stateManager;
}

@Override
public void Op(int CID, byte[] requests, MessageContext msgCtx) {

}

@Override
public void noOp(int CID, byte[][] operations, MessageContext[] msgCtx) {
    logger.debug("NoOp");
    //for (int i = 0; i < msgCtx.length; i++)
    //    logRequest(operations[i], msgCtx[i]);

    for (byte[] operation : operations) {
        Object obj = TOMUtil.getObject(operation);
        if (obj instanceof ReconfigureRequest) {
            logger.info("Reconfiguration");
            ReconfigureRequest reconfigureRequest = (ReconfigureRequest) obj;
            for (Integer key : reconfigureRequest.getProperties().keySet()) {
                String value = reconfigureRequest.getProperties().get(key);
                if (key == ServerViewController.CHANGE_F) {
                    int f = Integer.valueOf(value);
                    if (currentF < f) {
                        logger.info("Increasing f. {}->{}", currentF, f);
                    } else if (currentF > f) {
                        logger.info("Reducing f. {}->{}", currentF, f);
                    }
                    currentF = f;
                }
            }
        }
    }
}

@Override
public byte[] executeOrdered(byte[] command, MessageContext msgCtx) {
    Request request = preprocessRequest(command, msgCtx);
    if (request == null)
        return null;
    byte[] preprocessedCommand = request.serialize();
    byte[] response;
    if (request.getType() == MessageType.APPLICATION) {
        logger.debug("Received application ordered message of {} in CID {}. Regency: {}", msgCtx.getSender(),
                msgCtx.getConsensusId(), msgCtx.getRegency());
```

```
228             interServersCommunication.messageReceived(request.getPlainData(), msgCtx
    );
                response = new byte[0];
230         } else {
                stateLock.lock();
232             response = appExecuteOrdered(request.getPlainData(), request.getShares()
    , msgCtx).serialize();
                stateLock.unlock();
234         }
            logRequest(preprocessedCommand, msgCtx);
236
            return response;
238     }

240     @Override
        public byte[] executeUnordered(byte[] command, MessageContext msgCtx) {
242         Request request = preprocessRequest(command, msgCtx);
            if (request == null)
244             return null;
            if (request.getType() == MessageType.APPLICATION) {
246             logger.debug("Received application unordered message of {} in CID {}",
    msgCtx.getSender(), msgCtx.getConsensusId());
                interServersCommunication.messageReceived(request.getPlainData(), msgCtx
    );
248             return new byte[0];
            }
250
            return appExecuteUnordered(request.getPlainData(), request.getShares(),
    msgCtx).serialize();
252     }

254     public abstract ConfidentialMessage appExecuteOrdered(byte[] plainData,
    ConfidentialData[] shares, MessageContext msgCtx);

256     public abstract ConfidentialMessage appExecuteUnordered(byte[] plainData,
    ConfidentialData[] shares, MessageContext msgCtx);

258     public abstract ConfidentialSnapshot getConfidentialSnapshot();

260     public abstract void installConfidentialSnapshot(ConfidentialSnapshot snapshot);

262     private Request preprocessRequest(byte[] message, MessageContext msgCtx) {
            try (ByteArrayInputStream bis = new ByteArrayInputStream(message);
264              ObjectInput in = new ObjectInputStream(bis)) {
                MessageType type = MessageType.getMessageType(in.read());
266             Request result = null;
                int len;
268             byte[] plainData = null;
                switch (type) {
270                 case CLIENT:
                        len = in.readInt();
272                     if (len != -1) {
                            plainData = new byte[len];
274                         in.readFully(plainData);
                        }
276                     len = in.readInt();
                        ConfidentialData[] shares = null;
278                     if (len != -1) {
                            shares = new ConfidentialData[len];
280                         PrivatePublishedShares publishedShares;
                            for (int i = 0; i < len; i++) {
282                             publishedShares = new PrivatePublishedShares();
                                publishedShares.readExternal(in);
```

```
                            shares[i] = new ConfidentialData(confidentialityScheme.
extractShare(publishedShares));
                        }
                    }
                    result = new Request(type, plainData, shares);
                    break;
                case APPLICATION:
                    len = in.readInt();
                    plainData = new byte[len];
                    in.readFully(plainData);
                    result = new Request(type, plainData);
                    break;
            }
            return result;
        } catch (IOException | SecretSharingException e) {
            logger.warn("Failed to decompose request from {}", msgCtx.getSender(), e
);
            return null;
        }
    }

    private void saveState(byte[] snapshot, int lastCID) {
        logLock.lock();
        logger.debug("Saving state of CID {}", lastCID);

        log.newCheckpoint(snapshot, TOMUtil.computeHash(snapshot), lastCID);

        logLock.unlock();
        logger.debug("Finished saving state of CID {}", lastCID);
    }

    private void saveCommands(byte[][] commands, MessageContext[] msgCtx) {
        if (commands.length != msgCtx.length) {
            logger.debug("----SIZE OF COMMANDS AND MESSAGE CONTEXTS IS DIFFERENT----
");
            logger.debug("----COMMANDS: {}, CONTEXTS: {} ----", commands.length,
msgCtx.length);
        }
        logger.debug("Saving Commands of client {} with cid {}", msgCtx[0].getSender
(), msgCtx[0].getConsensusId());
        logLock.lock();

        int cid = msgCtx[0].getConsensusId();
        int batchStart = 0;
        for (int i = 0; i <= msgCtx.length; i++) {
            if (i == msgCtx.length) { // the batch command contains only one command
 or it is the last position of the array
                byte[][] batch = Arrays.copyOfRange(commands, batchStart, i);
                MessageContext[] batchMsgCtx = Arrays.copyOfRange(msgCtx, batchStart
, i);
                log.addMessageBatch(batch, batchMsgCtx, cid);
            } else {
                if (msgCtx[i].getConsensusId() > cid) { // saves commands when the
CID changes or when it is the last batch
                    byte[][] batch = Arrays.copyOfRange(commands, batchStart, i);
                    MessageContext[] batchMsgCtx = Arrays.copyOfRange(msgCtx,
batchStart, i);
                    log.addMessageBatch(batch, batchMsgCtx, cid);
                    cid = msgCtx[i].getConsensusId();
                    batchStart = i;
                }
            }
        }
```

```
338             logger.debug("Log size: " + log.getNumBatches());
                logLock.unlock();
340         }

342         private void logRequest(byte[] command, MessageContext msgCtx) {
                int cid = msgCtx.getConsensusId();
344             commands.add(command);
                msgContexts.add(msgCtx);
346
                if (!msgCtx.isLastInBatch()) {
348                 logger.debug("Not last in the batch");
                    return;
350             }

352             if (cid > 0 && (cid % checkpointPeriod) == 0) {
                    logger.info("Performing checkpoint for consensus " + cid);
354                 stateLock.lock();
                    ConfidentialSnapshot snapshot = getConfidentialSnapshot();
356                 stateLock.unlock();
                    saveState(snapshot.serialize(), cid);
358             } else {
                    saveCommands(commands.toArray(new byte[0][]), msgContexts.toArray(new
        MessageContext[0]));
360             }
                getStateManager().setLastCID(cid);
362             commands.clear();
                msgContexts.clear();
364         }
    }
```

Bibliography

[1] BFT-SMaRt. `http://bft-smart.github.io/library/`. Accessed on 20/04/2019.

[2] Yahoo! Cloud System Benchmark (YCSB). `https://github.com/brianfrankcooper/YCSB`. Accessed on 07/06/2019.

[3] Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K. Reiter, and Emin Gün Sirer. sAVSS: Scalable Asynchronous Verifiable Secret Sharing in BFT Protocols. *arXiv e-prints*, Jul 2018.

[4] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the Efficiency of Durable State Machine Replication. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 169–180. USENIX Association, 2013.

[5] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362. IEEE Computer Society, 2014.

[6] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. DepSpace: A Byzantine Fault-tolerant Coordination Service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 163–176, 2008.

[7] G.R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317. AFIPS Press, 1979.

[8] Ran Canetti, Shai Halevi, and Jonathan Katz. A Forward-secure Public-key Encryption Scheme. In *Proceedings of the 22Nd International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'03, pages 255–271. Springer-Verlag, 2003.

[9] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186. USENIX Association, 1999.

[10] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.*, 20:398–461, 11 2002.

[11] David Chaum. Blind Signatures for Untraceable Payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology*, pages 199–203, Boston, MA, 1983. Springer US.

[12] S. Duan and H. Zhang. Practical State Machine Replication with Confidentiality. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 187–196, Sept 2016.

[13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, April 1988.

[14] Paul Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 427–438, 1987.

[15] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, April 1985.

[16] Joni Silva Fraga and David Powell. A fault- and intrusion-tolerant file system. In *Proc. of the 3rd IFIP Conference on Computer Security*, 1985.

[17] Vassos Hadzilacos and Sam Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Cornell University, 1994.

[18] Lein Harn and Changlu Lin. Detection and identification of cheaters in (t, n) secret sharing scheme. *Designs Codes and Cryptography*, 52:15–24, 07 2009.

[19] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[20] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive Secret Sharing Or: How to Cope With Perpetual Leakage. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '95, pages 339–352, 1995.

[21] W. G. Horner. A New Method of Solving Numerical Equations of All Orders, by Continuous Approximation. *Philosophical Transactions of the Royal Society of London*, 109:308–335, 1819.

[22] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[23] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[24] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[25] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[26] M. Lepinski and S. Kent. RFC-5114. `https://www.rfc-editor.org/rfc/rfc5114.html`. Accessed on 18/07/2019.

[27] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[28] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct 1998.

[29] M. A. Marsh and F. B. Schneider. CODEX: a robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, Jan 2004.

[30] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17. ACM, 1988.

[31] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[32] Rafail Ostrovsky and Moti Yung. How to Withstand Mobile Virus Attacks (Extended Abstract). In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '91, pages 51–59, 1991.

[33] R. Padilha and F. Pedone. Belisarius: BFT Storage with Confidentiality. In *2011 IEEE 10th International Symposium on Network Computing and Applications*, pages 9–16, Aug 2011.

[34] Torben P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 129–140, 1992.

[35] Vassantlal Robin and Bessani Alysson. Confidencialidade em Serviços Tolerantes a Intrusão de Longa Duração. In *INForum2019*, 2019.

[36] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[37] Berry Schoenmakers. A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 148–164. Springer-Verlag, 1999.

[38] David Schultz, Barbara Liskov, and Moses Liskov. MPSS: Mobile Proactive Secret Sharing. *ACM Trans. Inf. Syst. Secur.*, 13(4):34:1–34:32, December 2010.

[39] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, November 1979.

[40] J. Sousa and A. Bessani. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48, May 2012.

[41] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, April 2009.

[42] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 253–267, 2003.

[43] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. COCA: A Secure Distributed Online Certification Authority. *ACM Trans. Comput. Syst.*, 20(4):329–368, November 2002.